MASTER EN SCIENCES INFORMATIQUES

Proposition d'une démarche de dérivation d'une application informatique à partir de ses spécifications fonctionnelles

Ruttens, Philippe

Award date:
1990

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

Proposition d'une démarche de dérivation d'une application informatique à partir de ses spécifications fonctionnelles

Philippe Ruttens

Promoteur : Professeur François Bodart

Mémoire présenté en vue de l'obtention
du titre de
Licencié et Maître en Informatique

Année académique 1989 - 1990

Mon premier souci sera d'exprimer mes plus vifs remerciements à mon promoteur, le Professeur François Bodart, ainsi qu'à mon maître de stage, le Professeur Yves Pigneur. Les remarques, critiques et conseils que tous deux m'ont prodigués au long de l'élaboration de ce mémoire ont permis de mener à bien ce travail, et ce dans les meilleures conditions possibles.

Je souhaite également remercier Gabor Maksay et Isabelle Petoud pour leur accueil et le temps qu'ils ont bien voulu me consacrer lors de mon stage à Lausanne.

Je désire exprimer ma profonde reconnaissance à Gilles Modart et Olivier Masson pour l'accueil et l'aide précieuse qu'ils m'ont bien gentiment fournies lors de la réalisation de mes travaux à Glaverbel (Bruxelles).

J'aimerais également exprimer toute ma gratitude envers les assistants de l'Institut d'Informatique et d'Organisation de l'Université de Lausanne ainsi qu'envers les assistants du Professeur Bodart pour leurs conseils toujours judicieux.

Qu'il me soit enfin permis de remercier tous ceux et celles qui ont , de près ou de loin, contribué à l'aboutissement de ce travail.

Résumé

Dans le cadre des nombreuses recherches visant à automatiser la conception d'applications informatiques de gestion, ce mémoire propose un contexte sémantique stable et un ensemble de techniques destinées à construire un processus industriel de réalisation automatique des solutions informatiques exploitables. Ce processus vise à dériver d'une façon continue et systématique une application informatique à partir de ses spécifications fonctionnelles exprimées selon un formalisme bien défini, et cela en ayant déterminé auparavant l'environnement cible choisi ainsi que les choix initiaux à poser. La réalisation des applications informatiques devrait dès lors se limiter à une activité de "calibrage" des systèmes automatiquement générés.

Ce processus se base sur un ensemble de modèles et est représenté par une démarche qui sera mise en oeuvre plus tard à l'aide d'outils logiciels.

Cependant, à l'heure actuelle, les limites d'application d'un tel processus sont relativement importantes et il faudra encore beaucoup de travail pour obtenir un résultat probant et adapté à l'évolution constante de l'informatique.

Ce travail présentera donc les concepts et techniques constitutifs de ce processus et étudiera l'application de ceux-ci dans 2 environnements différents. Il proposera également un maximum de suggestions à retenir pour la suite des recherches qui seront menées.

Abstract

As part of the numerous research works attempting to automate the design of management applications, this dissertation proposes a stable semantic context and a set of technologies aimed at the construction of an industrial method of automatic realization of computerized solutions. This method consists in deriving in a continuous and systematic way a computerized application from its functional specifications expressed in a well-defined formalism. This is done after having determined the physical target environment and the initial choices. Therefore, the realization of computerized applications should be restricted to a "gauging" activity of the automatically generated systems.

This method is based upon a set of models and is represented by a step by step process which will be backed up later by advanced software tools.

However, at present, the limits of application of such a theory are quite significant and much work will still be needed to obtain a suitable result adapted to the constant evolution of computer science.

This work will therefore present the concepts and mechanisms which are constituent of the proposed method and it will analyse their application in 2 different environments. It will also propose as many suggestions as possible for the continuation of research.

Table des matières

Table des figures IV

Introduction 1

Chapitre I - CONTEXTE DE BASE 3

I.1 La méthode IDA 3
I.2 Les modèles 5
I.3 Les outils logiciels 9
I.4 La démarche 9
I.5 Conclusion 10

Chapitre II - MODELES DE TRANSFORMATION 11

II.1 Spécifications fonctionnelles d'une phase 11
II.2 Module informatique 13
II.3 Machine-phase 16
II.4 Module - Machine abstraite 19
II.5 Langage de spécification 22
II.5.1 *Problématique* 22
II.5.2 *Langage DESPATH+* 24
II.5.2.1 Enumération et définition abrégée des primitives et mots-clés 25
II.5.2.2 Explication détaillée et illustrée des primitives 27
II.5.2.3 Exposé des structures de contrôle 31

II.6 Illustration : Machine-phase GestionClients 32
II.6.1 *Mémoire* 33
II.6.2 *Paramètres* 35
II.6.3 *Procédures* 36
II.6.4 *Actions dynamiques* 37

V.2.4 *Transformation de base des fonctions, messages et traitements* 96
V.2.5 *Application des techniques de simplification des paramètres* 97
V.2.6 *Application des techniques d'introduction d'une mémoire d'exécution* 98
V.2.7 *Production de l'interface détaillée de la machine-phase* 99
V.2.8 *Aplication de la technique de changement de représentation* 99
V.2.9 *Application de la technique d'utilisation de modules auxiliaires* 100
V.2.10 *Mise sous forme algorithmique* 101
V.2.11 *Codage en langage cible* 101
V.2.12 *Points de contrôle* 102
V.2.13 *Points de décision* 102

V.3 Conclusion 104

Chapitre VI - SUGGESTIONS POUR L'OUTIL LOGICIEL ET SUJETS DE REFLEXION 105

VI.1 Suggestions pour l'outil logiciel 105
VI.2 Sujets de réflexion 106
VI.2.1 *Mise en oeuvre de la démarche de dérivation* 106
VI.2.2 *Vue "Application - Machine-application"* 108
VI.2.3 *Changements de représentation de la base de données* 109
VI.2.4 *Critique du principe de la mémoire d'exécution* 110
VI.2.5 *Contexte organisationnel* 112

Conclusion 113

Bibliographie 115

Annexe A : Application développée en COBOL

Annexe B : Application développée en DELTA

Table des figures

Figure I.1 : Etapes du cycle de vie d'un S.I et domaine d'application d'IDA..... 4
Figure I.2 : Monenclature standard de structuration des traitements. 8
Figure I.3 : Représentation schématique d'une fonction..... 9
Figure II.1 : Démarche de construction d'une phase..... 12
Figure II.2 : Construction systématique d'une application informatique intéractive..... 13
Figure II.3 : Description d'un module informatique..... 16
Figure II.4 : Tableau récapitulatif de la transformation d'une phase en machine-phase..... 17
Figure III.1 : Illustration des différentes techniques mises en oeuvre dans la conception et l'implantation d'une machine-phase..... 58
Figure III.2 : Architecture globale d'une application informatique..... 61
Figure IV.1 : Choix de l'architecture d'implantation de la machine-phase en langage COBOL..... 66
Figure IV.2 : Utilisation des processeurs pour la gestion de transactions..... 75
Figure IV.3 : Zones de stockage de données en DELTA..... 76
Figure IV.4 : Structure des programmes en DELTA et en COBOL..... 78
Figure IV.5 : Squelette général des programmes DELTA..... 81
Figure IV.6 : Choix de l'architecture d'implantation de la machine-phase en DELTA 82
Figure IV.7 : Correspondance DELTA - Spécifications techniques 90
Figure V.1 : Proposition de démarche de dérivation..... 93
Figure VI.1 : Vue Application - Machine-application..... 109
Figure VI.2 : Mémoire d'exécution globale..... 111

INTRODUCTION

La conception des systèmes d'information et le développement des applications informatiques a suscité depuis longtemps un important effort de recherche et a donné lieu à l'élaboration de nombreuses méthodes afin d'aider le spécialiste dans sa tâche.

Dans le cadre de cette problématique, une attention toute particulière a toujours été portée à la construction et à l'expression des spécifications fonctionnelles, autrement dit la conception fonctionnelle.

" Cette étape de spécification est tenue pour essentielle puisqu'elle a pour objet de constituer une fondation solide au développement des applications informatiques. La qualité d'un système informatique opérationnel dépend en effet en grande partie de la rigueur apportée à sa conception initiale et de l'adéquation de la solution retenue aux objectifs de l'organisation. " [BODART-PIGNEUR,89].

Il est donc très important d'exprimer correctement les besoins des futurs utilisateurs de la solution informatique et cela aussi bien dans le domaine de la recherche que dans le monde industriel et commercial.

Les spécifications fonctionnelles qui permettent de remplir cet objectif sont en fait regroupées en trois grandes catégories : les spécifications exprimées de façon informelle (en langue naturelle par exemple), les spécifications exprimées de façon formelle (spécifications algébriques, types abstraits,....) et les spécifications exprimées de façon mixte. Tous ces types de spécifications devront bien sûr être indépendants de tout moyen de réalisation de l'application.

Mais au-delà de ce problème d'expression une approche résolument moderne du développement des systèmes d'information est prônée. En effet, une des priorités majeures de la recherche actuelle consiste à élaborer une démarche ou une méthode permettant de passer directement de ces spécifications à l'application informatique finale. La nécessité d'une systématique accrue dans la réalisation des applications informatiques devient ainsi peu à peu le fil conducteur des travaux entrepris. Dans ce contexte, la réalisation d'une application devra donc idéalement être dérivée d'une façon continue et systématique de son analyse fonctionnelle (cette étape couvrant en général l'étude d'opportunité et l'analyse conceptuelle).

Cette idée de transformation quasi-automatique des spécifications fonctionnelles conduit tout naturellement au concept de spécifications exécutables qui représente le but idéal pour les informaticiens de tous domaines.

Cet objectif constitue notamment l'un des domaines de recherche du professeur Yves Pigneur de l'Université de Lausanne qui propose un processus systématique et continu de dérivation du code exécutable à partir des spécifications fonctionnelles d'une application informatique.

Ce processus de dérivation, qui représente une des solutions possibles à la problématique évoquée, constituera la pierre angulaire de ce mémoire et l'ensemble des concepts et méthodes mises au point par les professeurs Yves Pigneur et François Bodart seront expliqués en détail dans la première partie de ce travail.

Ce mémoire se propose d'atteindre en fait plusieurs objectifs complémentaires .

Le premier objectif est d'exposer, à partir d'un contexte sémantique stable et bien défini, l'ensemble des concepts et techniques proposées dans le cadre de ce processus de dérivation quasi-automatique d'une application informatique.

Le **deuxième objectif** est d'énoncer les problèmes liés à la mise en place des transformations à faire pour atteindre le but désiré. Les chapitres I, II, III et VI sont destinés à atteindre ces deux premiers objectifs.

Le **troisième objectif** est d'appliquer dans deux environnements bien distincts les concepts et méthodes étudiées. Le premier contexte est constitué par le langage COBOL, le langage relationnel SQL-DS et le gestionnaire d'interface ISPF. Le deuxième contexte est constitué par le générateur de code et de transactions DELTA. Cet objectif sera atteint par le chapitre IV.

Le **quatrième objectif** vise à proposer une démarche de dérivation de l'application informatique ou plutôt d'une partie de celle-ci, démarche élaborée à partir des concepts vus. Cela sera exposé dans le chapitre V.

Enfin, le **cinquième objectif** sera de proposer quelques suggestions concernant l'outil logiciel destiné à supporter la démarche construite. Ce support mettra à disposition du programmeur, dans un environnement donné et avec un langage cible donné, un processus industriel de construction quasi-automatique de l'application informatique . Cet objectif sera atteint par le chapitre VI.

Ce mémoire n'a donc aucune autre prétention que de situer le problème de la dérivation automatique d'une application informatique à partir de ses spécifications fonctionnelles en se basant sur un contexte sémantique donné et en se limitant à celui-ci.

A l'heure actuelle, les techniques informatiques sont encore insuffisantes pour envisager, telle quelle et à court terme, la transformation des spécifications en applications opérationnelles, optimisées et exactement adaptées aux besoins et il est malheureusement très difficile, voire impossible de proposer une méthode ou une démarche précise et générale réglant ce problème de façon absolue.

CHAPITRE I - CONTEXTE DE BASE

Ce chapitre a pour but d'exposer le contexte sémantique utilisé en tant que base de la démarche de dérivation de l'application informatique. Ce contexte présente une des méthodes préconisées pour exprimer et gérer les spécifications fonctionnelles d'un système d'information (S.I).

I.1 La méthode IDA

Le contexte sémantique sur lequel nous nous baserons est constitué par la méthode IDA (*Interactive Design Approach*) développée par les Professeurs François Bodart (F.U.N.D.P - Namur) et Yves Pigneur (Université de Lausanne). Nous ne présenterons ici qu'un résumé des concepts de l'environnement IDA que nous utiliserons pour la suite du travail. Ce résumé sera constitué d'extraits du livre exposant en détail toute cette théorie [BODART-PIGNEUR, 89]. La totalité de ce chapitre I est donc composée de phrases et figures reprises littéralement de ce livre (pour des raisons de présentation, l'emploi de guillemets fut évité dans ce chapitre).

La méthode d'informatisation IDA englobe en fait des modèles, une méthode et des outils permettant d'assister la conception des systèmes d'information de gestion. Mais resituons d'abord le domaine d'application de la méthode IDA dans le cycle de vie d'un S.I .

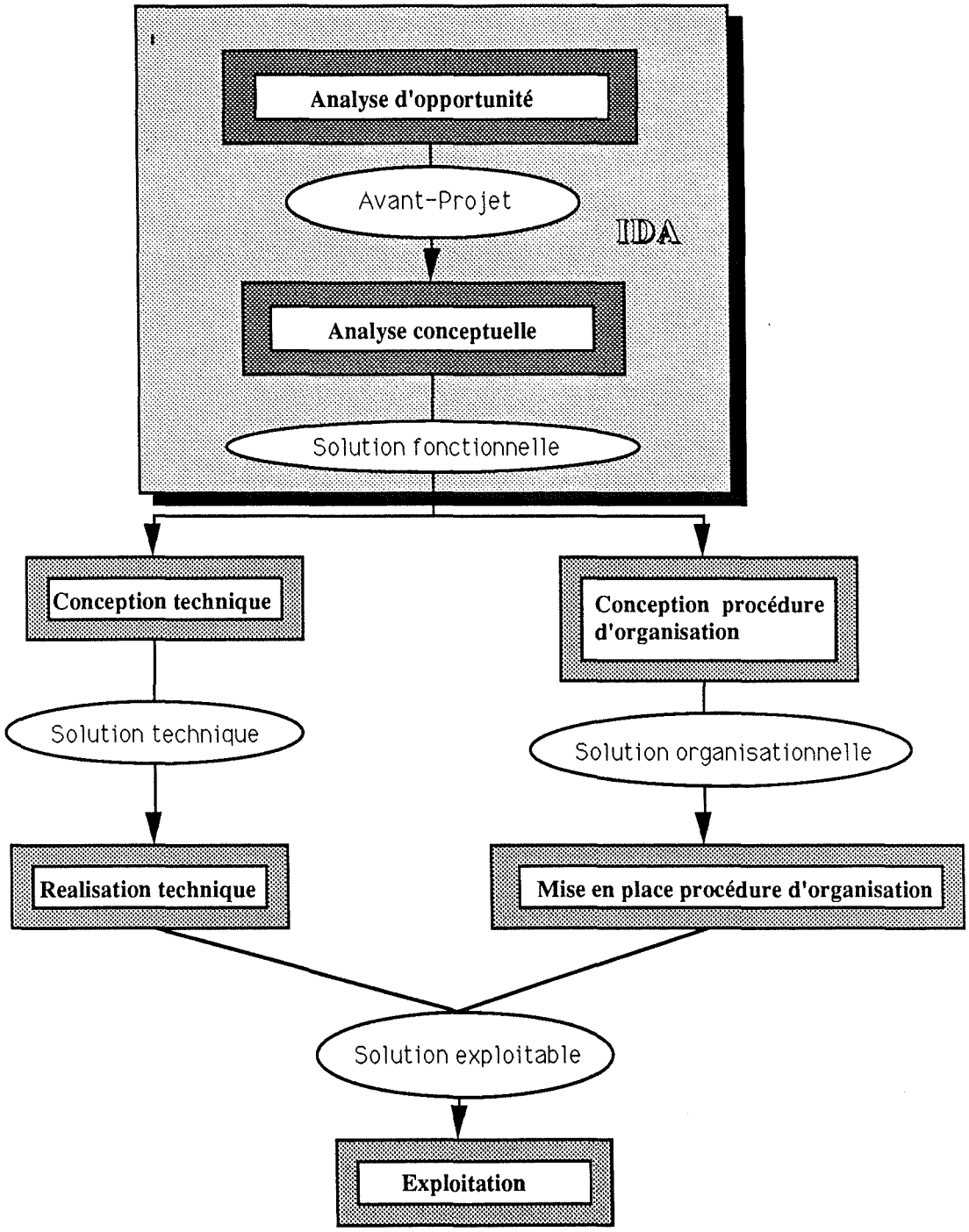


Figure I.1 - Etapes du cycle de vie d'un S.I et domaine d'application d' IDA

La figure I.1 nous montre les étapes du cycle de vie d'un S.I. IDA s'applique en fait aux deux premières étapes de celui-ci à savoir l'analyse d'opportunité et l'analyse conceptuelle. Rappelons brièvement le principe de ces 2 étapes :

L'**analyse ou étude d'opportunité** prépare un avant-projet de solution à partir des besoins exprimés par l'organisation, cette étape s'inscrivant idéalement dans le cadre du schéma directeur des S.I de l'organisation.

L'**analyse conceptuelle** élabore sur base de l'avant-projet une solution fonctionnelle détaillée mais indépendante de tout moyen de réalisation.

L'idée majeure d'IDA est de penser le développement d'un S.I comme un travail progressif de spécification par affinements successifs et prise en compte séquentielle des contraintes sans retour en arrière. Au lieu de concevoir d'emblée les S.I à travers les contraintes spécifiques de moyens informatiques contingents, IDA s'attarde à une spécification conceptuelle, strictement fonctionnelle, et indépendante des solutions informatiques éventuelles.

Les spécifications fonctionnelles sont exprimées dans un langage formel de haut niveau (DSL). Ce langage constitue un univers logique strict et homogène, et correspond, comme nous allons le voir, à un ensemble de modèles riches dont la sémantique, analysable par ordinateur, est bien définie. Ce langage formel de haut niveau garantit la pertinence des S.I à l'aide de tests puissants au niveau de la spécification fonctionnelle elle-même.

La méthode IDA est une méthode de développement des S.I qui propose une **démarche** fondée sur des **modèles** et mise en oeuvre à l'aide d'**outils logiciels**.

Examinons brièvement les éléments qui nous intéressent dans chacune de ces 3 parties .

I.2 Les modèles

Chaque modèle est formé de concepts et de règles relatives à leur utilisation et se rapporte à un aspect particulier du S.I à élaborer.

Les 6 modèles proposés sont :

1 . Le **Modèle Entité-Association (E/A)**

2 . Le **Modèle de structuration des traitements :** Ce modèle est crucial pour la compréhension de la suite du mémoire. Le modèle de structuration des traitements comprend un ensemble de concepts et de règles destinées à structurer un S.I en une hiérarchie d'agrégats fonctionnels de traitements. Un agrégat comprend un ensemble de règles à suivre ou d'actions à entreprendre pour réaliser une fonctionnalité du S.I. A chaque agrégat seront associés la définition et l'objectif de cette fonctionnalité.

Le modèle de structuration des traitements est basé sur la décomposition sous forme arborescente où l'on établit entre les traitements une relation de partition : tout traitement, sauf le traitement initial, fait partie d'un traitement de niveau supérieur. Cette relation de structuration exprime un objectif de description statique des traitements d'un S.I.

Pour chaque traitement, on spécifie son nom, sa définition et les objectifs qui lui sont assignés, son niveau dans la structure de décomposition, les relations avec les autres traitements et la performance fonctionnelle souhaitée.

Dans cette décomposition arborescente, on distingue 4 niveaux ou repères privilégiés, compte tenu de leur signification particulière. Ces repères constituant une monenclature standard sont représentés par la Figure I.2.

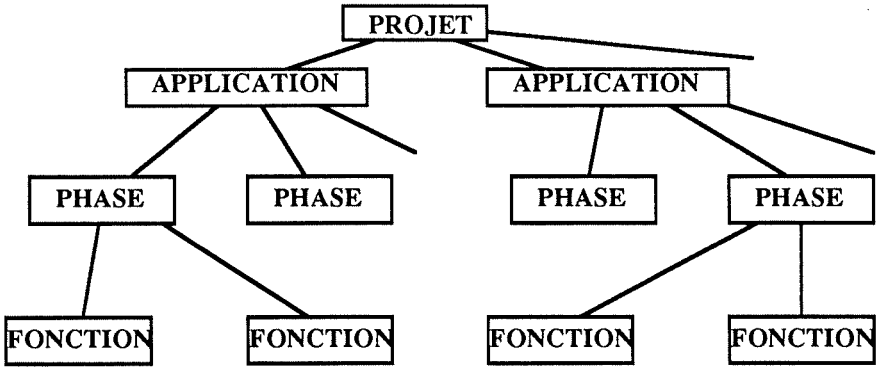


Figure I.2 - Monenclature standard de structuration des traitements

Quoique la phase constitue le repère central de la monenclature, pour des raisons de cohérence méthodologique, ces concepts sont présentés suivant l'ordre descendant. Chacun de ces niveaux possède des critères d'identification propres mais limitons-nous à leur définition :

- Un PROJET est la partie du S.I qui fait l'objet d'une analyse. C'est un sous-système du S.I à réaliser.

- Une APPLICATION est un traitement quasi-autonome par rapport aux autres applications d'un projet : elle constitue une unité de planning dans la gestion d'un projet. Une application est la plus petite partie d'un projet dont le cycle de développement doit être considéré globalement : l'analyse d'implémentation, la réalisation et l'exploitation se rapporteront aux spécifications fonctionnelles d'une application au minimum.

- Une PHASE est un traitement possédant une unité spatio-temporelle d'exécution. Cette unité d'exécution implique que la phase soit entièrement exécutée dans une cellule d'activités, c'est à dire un centre d'activité homogène dans le temps et dans l'espace, doté de ressources humaines et/ou matérielles et pourvu de règles de comportement nécessaires à son fonctionnement.

Le concept de phase constitue le repère central de la monenclature par la signification qu'il présente sur 3 plans d'analyse :

Au plan informationnel, la phase est un lieu d'identification de structures homogènes de données et de règles de traitement.

Elle constitue un lieu charnière de la spécification d'une solution conceptuelle : on travaillera d'abord au niveau de la phase et l'on procèdera ensuite à la consolidation au niveau de l'application. Sur ce plan, le concept de phase est particulièrement significatif pour l'informaticien.

Au plan économique, elle est un lieu d'allocation des ressources humaines,matérielles et/ou logicielles, nécessaires à son exécution.

Au plan organisationnel, elle est un lieu de redéfinition des structures d'organisation : tâches, fonctions, rôles, responsabilités, postes de travail.

Aux plans économique et organisationnel, le concept de phase est totalement significatif pour le second profil de spécialisation impliqué dans une analyse fonctionnelle, à savoir l'organisateur. Elle constitue, en effet, le lieu d'analyse des choix d'investissements élémentaires impliqués par l'évolution d'un S.I.

Pour l'utilisateur, troisième partenaire impliqué dans l'analyse d'un S.I, la phase identifie le cadre de référence de son poste de travail.

La phase constitue ainsi le lieu élémentaire d'analyse des changements à apporter à un S.I : changement des règles de mémorisation et de traitement, modification du comportement des personnes, changement dans les structures d'organisation, changement technologique.

Ce concept primordial servira de base à tout le travail présenté dans la suite de ce mémoire.

- Une FONCTION correspond au niveau élémentaire de la monenclature des traitements; elle résulte de la décomposition d'une phase en sous-traitements. Elle est associée à un objectif et un comportement considérés comme élémentaires par l'organisation, l'utilisateur et le concepteur du S.I.

Pour l'organisateur, elle traduit un objectif de gestion élémentaire assigné au S.I. Pour l'utilisateur final, elle représente un échange indécomposable de messages entre lui et le S.I. Par échange indécomposable, on entend un traitement, sans intéraction, qui, à partir d'informations fournies par l'utilisateur, lui en restitue d'autres après un certain temps. Un message est en provenance ou à destination soit de l'environnement du S.I soit d'un autre traitement. A une fonction doivent donc être associés les messages élémentaires, et leurs propriétés, qu'elle reçoit en entrée et produit en sortie.

Pour le concepteur, la fonction correspond à une action minimale et cohérente sur les informations mémorisées. Par action minimale et cohérente, il faut comprendre un accès localisé à la mémoire du S.I qui en respecte l'intégrité ainsi que les propriétés définies dans le schéma conceptuel.

La figure I.3 donne une représentation schématique de la fonction qui met en évidence les trois aspects de sa définition.

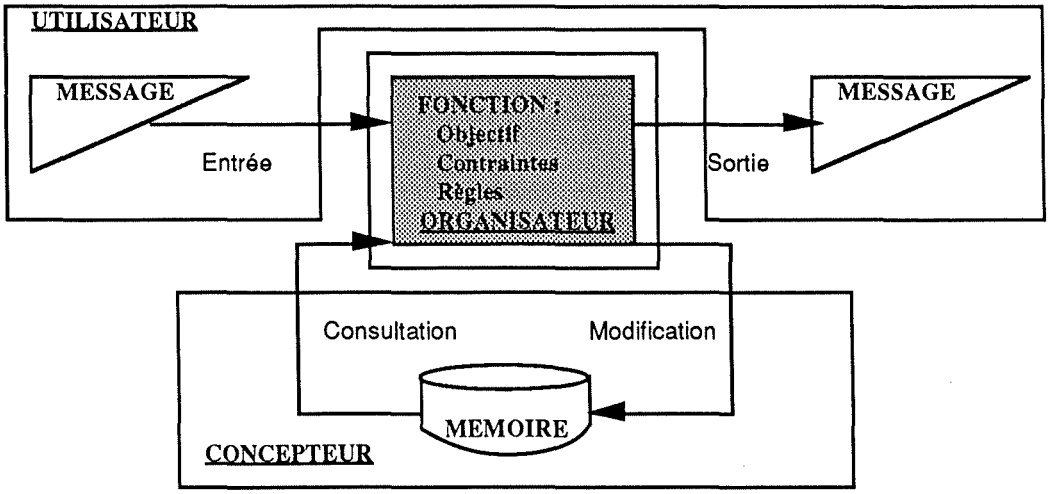


Figure I.3 - Représentation schématique d'une fonction

3 . Le **Modèle de la statique des traitements** est une application simple du modèle général d'un S.I permettant d'affiner l'idée de solution qu'un analyste se crée pour réaliser un projet, jusqu'à l'obtention d'une solution.

On procède à la spécification de la statique des traitements en deux étapes :

- pour un traitement d'un niveau donné de décomposition (souvent de niveau fonction), on le spécifie sous la forme d'un modèle de "boîte noire", c'est à dire qu'on en définit les informations en entrée et en sortie, les objectifs à réaliser par ce traitement, les actions primitives qui caractérisent les relations entre ces informations et le traitement (réception d'un message, consultation ou mise à jour de la mémoire du S.I,...) et les performances fonctionnelles souhaitées. Ce modèle de la boîte noire complète en fait la première spécification d'un traitement fournie par le modèle de structuration des traitements.
- on spécifie ensuite les règles de transformation des informations d'entrée (messages, mémoire du S.I) en informations de sortie (messages, mémoire du S.I modifié) c'est à dire les règles de traitement (contrôles de validité, calculs, ...)

4 . Le **Modèle de la dynamique des traitements** est principalement destiné à représenter, et donc à spécifier, les conditions d'exécution et d'enchaînement des traitements en vue de caractériser le comportement d'un S.I.

5 . Le **Modèle des ressources**

6 . Le **Modèle des flux de messages (diagramme des flux)**

L'exposé de ces modèles sera complété dans IDA par l'énoncé et l'analyse de règles de validation destinées à vérifier si les spécifications obtenues par l'application d'un modèle donné sont correctes : d'une part la complétude de chaque modèle sera vérifiée, c'est à dire que l'on vérifiera si chaque classe d'objet reprise dans la spécification possède bien, en fonction de son type, toutes les propriétés prévues dans le modèle sur lequel la spécification est basée; d'autre part, la cohérence des modèles sera testée c'est à dire que l'on vérifiera si les spécifications des différentes classes d'objets d'un schéma ne sont ni contradictoires entre elles, ni contradictoires avec les classes d'objets d'autres schémas (pas de redondances, contradictions ou ambiguités).

I.3. Les outils logiciels

L'emploi d'outils automatisés peut aider efficacement l'analyste à vérifier si le schéma conceptuel qu'il construit est communicable, complet, cohérent, réalisable et conforme aux besoins.

Le recours à des outils automatisés est d'autant plus indispensable qu'un objectif à terme est d'automatiser les étapes de réalisation de la solution conceptuelle retenue.

I.4 La démarche

La démarche, ce troisième pôle de toute méthodologie de conception, doit être vue comme un ensemble de règles et de propositions générales qui précisent notamment comment mettre en oeuvre les modèles et les outils automatisés en vue de maîtriser les étapes de l'étude d'opportunité et de l'analyse conceptuelle.

La démarche d'IDA possède un caractère générique, c'est un cadre adaptable au contexte particulier d'une organisation et d'un projet. L'ossature proposée devra donc être personnalisée en fonction du contenu du projet à développer et d'autres facteurs.

Une démarche minutieuse et générale semble illusoire et serait contraire aux objectifs recherchés par IDA, à savoir favoriser le travail créatif et la productivité des analystes.

I.5 Conclusion

Cette démarche bien définie, cet environnement logiciel, cette formalisation et cette lisibilité des spécifications fonctionnelles devraient, nous l'espérons, remplir la double exigence de cohérence et d'adéquation des spécifications aux objectifs de l'organisation ainsi que celle d'une participation accrue des gestionnaires au développement des S.I.

IDA permet d'engendrer des spécifications fonctionnelles possédant 3 types de qualités fondamentales : formalisées et lisibles, vérifiables et expérimentables, incrémentales.

Mais même si elles possèdent ces qualités, les spécifications fonctionnelles ne constituent pas une fin en soi : elles ne garantissent pas à elles seules la qualité des applications informatiques. Il faudra apporter la même rigueur et la même systématique à la réalisation des applications informatiques (en ce qui nous concerne : les étapes de conception technique et réalisation technique du cycle de vie d'un S.I) afin de profiter au maximum du soin apporté à la conception des spécifications fonctionnelles.

En d'autres termes, il est hautement souhaitable que les applications informatiques soient le plus directement possible issues ou dérivées des spécifications fonctionnelles. Cette dérivation directe devrait contribuer de façon significative à la qualité des S.I opérationnels et à la productivité de leur développement , à fortiori de leur maintenance.

CHAPITRE II - MODELES DE TRANSFORMATION

Ce chapitre a pour but de décrire les modèles et concepts qui sont à la base du processus de dérivation de l'application informatique à partir de ses spécifications fonctionnelles.

Le contexte sémantique de départ étant bien défini (cfr chapitre I), la partie de l'environnement IDA qui nous intéresse le plus pour le moment est constituée par les différents modèles de représentation d'un S.I et plus particulièrement les modèles de structuration des traitements et de la statique des traitements.

En effet, la découpe d'une application informatique en agrégats de traitements de niveau Phase ou de niveau Fonction est fondamentale car elle est à la base de la démarche de transformation des spécifications fonctionnelles. Comme nous allons le voir, le concept de phase servira directement à élaborer d'autres modèles qui la représenteront et permettront, grâce à diverses techniques, d'obtenir progressivement le résultat désiré, à savoir l'application informatique prête à être implantée.

La première transformation qui est envisagée est le passage des spécifications fonctionnelles d'une phase (telle que vues dans la méthode IDA) aux spécifications techniques d'un module informatique et plus précisément d'une machine-phase.

Mais avant d'expliquer ces nouvelles notions rappelons, en les détaillant, les principes de base des spécifications fonctionnelles de la phase qui constitue le repère central de la monenclature des traitements d'un S.I.

II.1 Spécifications fonctionnelles d'une phase

" Comme tout traitement, une phase est définie par ses objectifs , les performances fonctionnelles souhaitées, les informations en entrée et en sortie, les actions primitives caractérisant les relations entre les informations et les traitements ainsi que les règles de transformation des informations d'entrée en informations de sortie. " [BODART-PIGNEUR,89].

Une phase est décomposée en fonctions correspondant chacune à un traitement élémentaire significatif pour l'organisation, pour l'utilisation et pour la conception.

Cette décomposition en fonctions est guidée par les objectifs de la phase et par sa structuration de données (mémoire du S.I). Les fonctions sont identifiées et décrites selon une démarche itérative, déductive et guidée par les informations. Cela signifie en clair que l'on part des résultats (effets) de la phase pour dégager de proche en proche les fonctions en se rapprochant des arguments (causes) de la phase. Le processus se termine quand les arguments des fonctions correspondent à ceux de la phase.

Pour chaque fonction, on précisera ses messages reçus et générés (données et contraintes) et on décrira son comportement interne (c'est à dire ses actions sur les informations et ses règles de traitement).

Illustrons cette démarche à l'aide du schéma présenté dans la figure II.1.

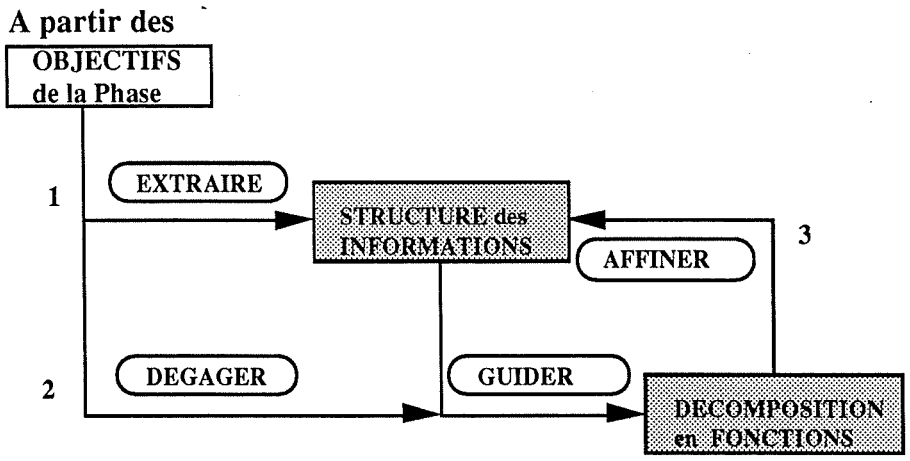


Figure II.1 - Démarche de construction d'une phase

A partir des objectifs de la phase, on extrait la structure des informations nécessaires et on dégagera, en étant guidé par cette structure d'informations, la décomposition en fonctions qui permettra d'affiner éventuellement les informations stockées et traitées.

Enfin, le schéma de la dynamique des fonctions de la phase décrira le comportement global de celle-ci et notamment l'enchaînement de ses fonctions.

Introduisons à présent la nouvelle problématique .

II.2 Module informatique

La construction systématique d'une application informatique intéractive peut être schématisée comme suit [PIGNEUR,90] :

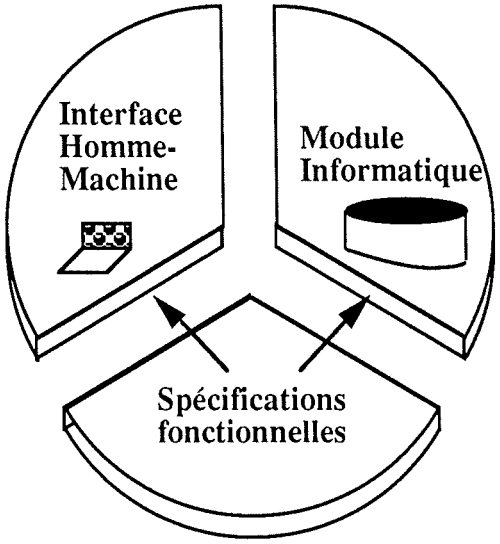


Figure II.2 - Construction systématique d'une application informatique intéractive

Les spécifications fonctionnelles d'une application intéractive donnent lieu à la construction d'une interface Homme-Machine et d'un module informatique. Ces deux parties devront pouvoir être développées indépendamment l'une de l'autre.

La notion d'interface Homme-Machine (appelée aussi dialogue avec l'utilisateur) dérivée à partir des spécifications fonctionnelles n'est pas l'objet de ce travail et nous ne détaillerons donc pas cette partie. Les travaux de recherche d'Isabelle Petoud (Université de Lausanne) sur ce sujet peuvent être consultés pour plus d'informations , notamment [PETOUD,88].

La notion de module informatique peut être expliquée d'abord d'une façon simpliste : cette partie est constituée par tous les traitements de l'application et notamment par les traitements de gestion de la base de données, ainsi que par cette base de données.

Afin d'expliquer clairement ce qu'est un module informatique et ses relations avec l'interface Homme-Machine, il est bon d'établir un parallélisme avec le modèle qui est à son origine à savoir le modèle classique de la machine à calculer [PIGNEUR,89] .

Une machine à calculer est composée également de 2 parties bien disjointes :

- une partie interface constituée comme suit :
  1. Mémoire d'affichage
  2. Scénario ou enchaînement d'opérations (conversation)
  3. Opérations de saisie ou d'affichage d'un nombre (présentation)
  4. Opérations de mise sous tension ou de fermeture de la machine ou de la mémoire d'affichage.

- une partie application constituée comme suit :
  1. Etat (mémoire de nombres)
  2. Opérations (addition, multiplication,...)
  3. Type et domaine des nombres
  4. Actions dynamiques (allumer, éteindre, nettoyer la mémoire,...)

  Tous les éléments de cette partie seront visibles par la partie interface sauf l'état qui est caché à celle-ci.

Un module informatique est très similaire à la partie application de ce modèle de machine à calculer.
Un module informatique est en effet constitué de 4 éléments :

1. **Mémoire** (ou Etat) représentée par un schéma E/A et ses contraintes d'intégrité (invariant)
2. **Fonctions** (Opérations) qui modifieront ou consulteront la mémoire
3. **Messages** (type, nom et domaine) qui correspondent aux arguments (messages reçus) et résultats (messages générés) des fonctions
4. **Actions dynamiques** destinées à installer, désinstaller, activer ou désactiver le module informatique

Signalons que ce dernier élément nous intéressera moins pour la suite du processus de dérivation. Ces actions dynamiques servent simplement à assurer les changements d'états (vus en tant que situations de fonctionnement) du module informatique, ces états étant : inopérande (=non-installée), installée (=non-active) et active (=exécution). Si on fait une analogie avec les systèmes gestionnaires de bases de données, l'installation/désinstallation correspond à la connexion/déconnexion avec la base de données et l'activation/désactivation correspond à l'ouverture et à la fermeture d'une transaction.

Un module informatique est donc constitué à la fois d'objets et de traitements.

Ajoutons qu'une obligation de preuve existe concernant chaque module informatique : il s'agit en effet de garantir la préservation de l'invariant de la mémoire du module à tout moment de son exécution. Chaque fonction et action dynamique devra respecter la structure de la mémoire et toutes ses contraintes d'intégrité.

Nous aurons donc pour chaque fonction l'assertion suivante :

{Invariant & Précondition} Fonction {Invariant},

Etant donné l'invariant portant sur la mémoire du module et la précondition de la fonction, après exécution du corps de la fonction, l'invariant de la mémoire doit rester vérifié.

Un module informatique possède une partie VISIBLE de l'extérieur (notamment par d'autres modules informatiques) qui est composée de la signature des fonctions (c'est à dire la déclaration, l'entête de cette fonction sous une forme choisie), des actions dynamiques et des messages échangés.

La partie CACHEE du module est composée de sa mémoire et du corps de ses différentes fonctions. Les caractéristiques de cette partie ne sont donc connues que par le module lui-même. Les personnes utilisant les fonctions du module informatique ne devront donc pas se soucier de la façon dont sont implémentées les fonctions et les informations stockées et traitées par le module.

Pour utiliser les fonctions du module, il suffira simplement de faire appel à ces fonctions (en se référant à leur signature) et d'échanger les messages adéquats.

Une relation UTILISE existe entre les modules informatiques, cette relation est définie comme suit :

Soit les deux modules informatiques M1 et M2,

M1 UTILISE M2 <==> . M1 fait appel à une ou plusieurs fonctions de M2 (exécution des traitements et échange de messages)
. M1 est incorrect si M2 ne fonctionne pas et on suppose que M2 fonctionne
. M1 se sert des spécifications de M2 pour effectuer les appels de fonction

Toutes ces notions sont résumées ci-dessous dans le schéma de la figure II.3.

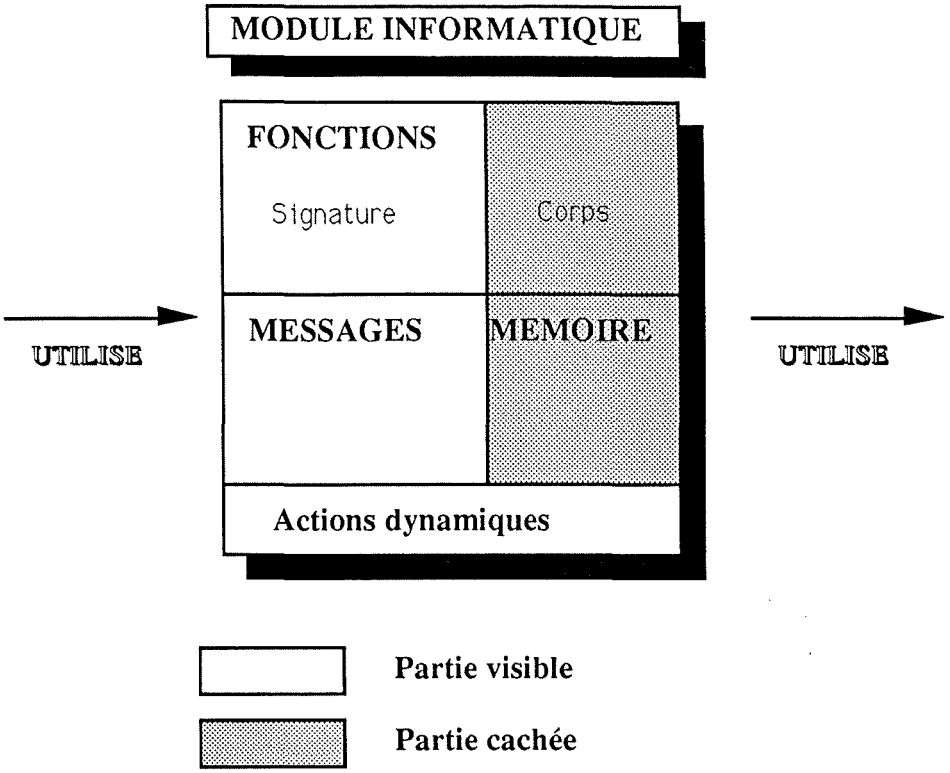


Figure II.3 - Description d'un module informatique

II.3 Machine-phase

Notre objectif principal est donc d'obtenir une dérivation continue de l'architecture d'une application informatique à partir de ses spécifications fonctionnelles. A cette fin nous allons utiliser le concept de module informatique et l'appliquer au concept central défini dans les spécifications fonctionnelles à savoir la phase. Nous aurons donc une transformation de base dont l'origine est une phase et la cible une machine-phase.

Nous allons créer une machine-phase (appelée aussi machine fonctionnelle) qui représentera les spécifications techniques d'une phase et qui servira de lieu d'analyse dans le cadre de la transformation souhaitée. Les composants de cette machine-phase se verront appliquer ultérieurement d'autres transformations qui permettront l'obtention de l'application informatique voulue.

Le concept de machine-phase doit être défini le plus précisément possible : une machine-phase est un module informatique utilisable par d'autres composants de l'architecture d'une application informatique et qui correspond à une phase IDA (c'est à dire représentant les spécifications fonctionnelles d'une phase vue selon la méthode IDA).

Elle est composée d'une mémoire (la "base de données" de la machine-phase), de procédures (vues comme des services), de paramètres (arguments et résultats de ces procédures) et secondairement d'actions dynamiques opérant sur la situation de cette machine.

La machine-phase est donc le modèle de correspondance par rapport à la phase.

Précisons à l'aide du tableau de la Figure II.4 comment s'opère cette transformation de base (cfr [PIGNEUR,89]) , et prouvons que le concept de module informatique s'adapte parfaitement au concept de phase et assure pleinement l'objectif de continuité de la dérivation.

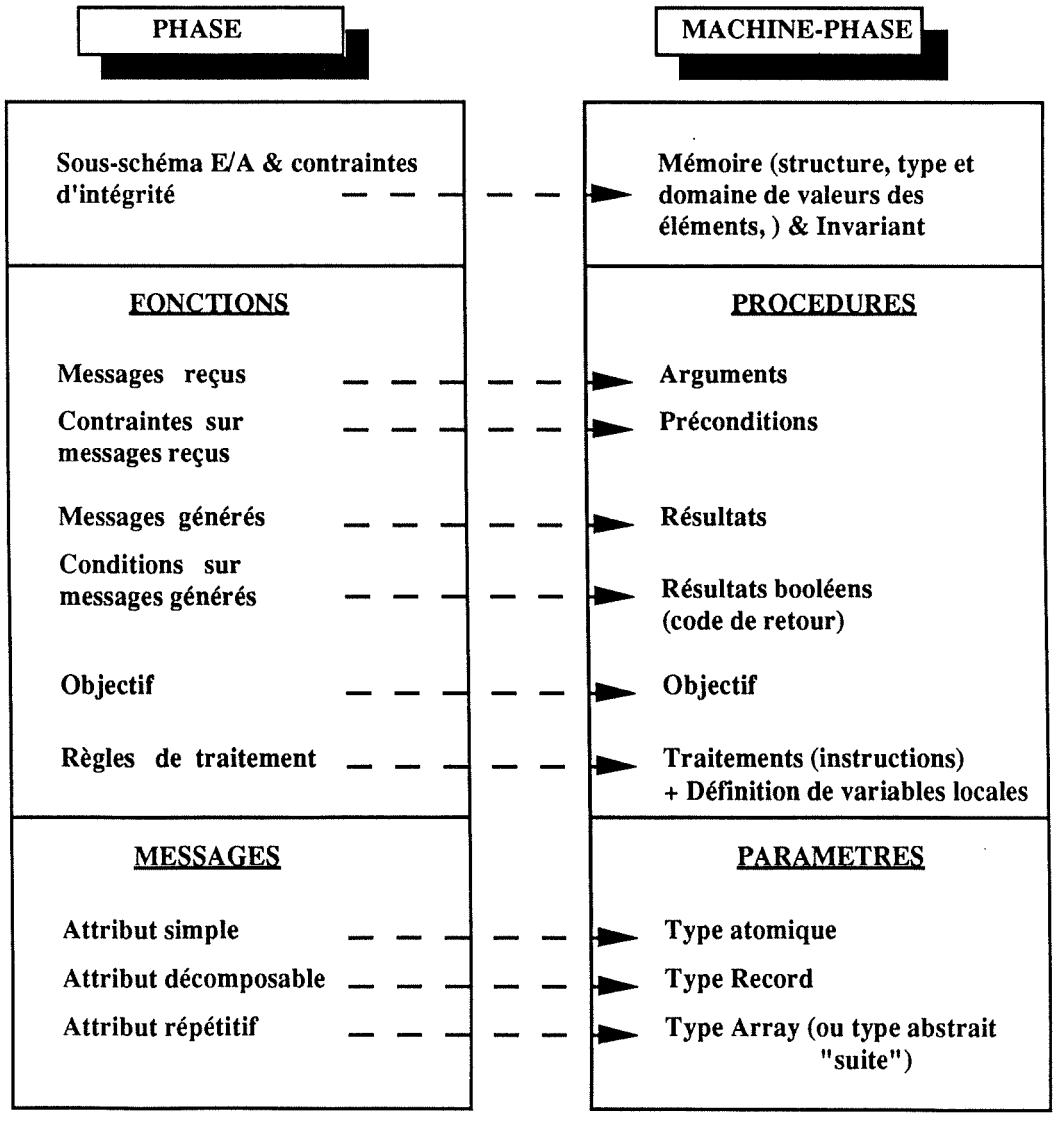


Figure II.4 - Tableau récapitulatif de la transformation d'une phase en machine-phase

Le sous-schéma E/A de la phase est transformé en mémoire de la machine-phase qui sera exprimée selon le modèle E/A ou éventuellement un autre formalisme. Le principe à respecter est le suivant : la mémoire doit être décrite de telle sorte que ce soit une description logique des fichiers de l'environnement cible et le plus souvent une représentation E/A du modèle cible.

Chaque fonction de la phase sera représentée par une procédure de la machine-phase et la signature (entête) de ces procédures sera exprimée selon le langage cible choisi pour l'implantation de l'application informatique (COBOL,PASCAL,...).

Chaque message reçu (généré) par une fonction sera transformé en un argument (résultat) dont la signature (forme) sera déterminée par le langage cible.

Les contraintes portant sur les messages reçus seront transformées en préconditions qui devront être vérifiées préalablement à l'exécution des procédures concernées.

Les conditions portant sur les messages générés donneront lieu à des résultats sous forme booléenne.

L'objectif de la machine-phase et de ses procédures sera évidemment strictement similaire à celui de la phase et de ses fonctions.

Les règles de traitement de la phase seront transformées en traitements décrits selon un langage semi-naturel dont l'objectif principal sera la compréhensibilité et la généralité. Nous étudierons ce langage à la fin de ce chapitre.

Les messages de la phase, selon leur type, seront traduits en paramètres d'un certain type (voir Figure II.4).

Notons que les termes *fonctions* et *messages* ont été transformés en *procédures* et *paramètres* afin d'éviter les ambiguités avec les concepts des spécifications fonctionnelles.

L'avantage de cette transformation est son **caractère systématique, simple et continu.** En effet, cette transformation respecte parfaitement la volonté de continuité de la démarche de dérivation en ne modifiant pas fondamentalement la structure (et bien sûr la sémantique) d'une phase mais en exprimant d'une nouvelle façon ses éléments de façon à obtenir progressivement une version exécutable des spécifications fonctionnelles.

Nous appliquerons par la suite plusieurs autres types de transformations (exposées au chapitre 2) à cette machine-phase, transformations qui permettront d'obtenir une machine physique qui sera l'implémentation, dans l'environnement cible choisi, des spécifications fonctionnelles de la phase de départ.

II.4 Module - Machine abstraite

Analysons à présent ce concept de machine-phase en le comparant aux concepts connus de module d'une part et de machine abstraite d'autre part.

Examinons d'abord le concept de **module** et effectuons une comparaison avec la machine-phase .

Selon D. Parnas, " on préfère souvent voir les systèmes informatiques divisés en *modules* . Cette division définit une relation FAIT PARTIE DE dont le graphe ne comporte jamais de boucles.
Un groupe de sous-programmes est rassemblé dans un module, les groupes de modules sont rassemblés en plus gros modules et ainsi de suite.
Notons que l'on peut permettre aux programmes d'un module d'appeler des programmes dans un autre module de telle sorte que la hiérarchie des modules définie ici n'a pas nécessairement de connexion avec la hiérarchie des programmes " [PARNAS, 72a].

Les procédures de la machine-phase peuvent être considérées comme des "sous-programmes" qui "font partie du" module constitué alors par la machine-phase . Mais on ne peut pas du tout assimiler la division du système en machines-phase avec une division en modules. Bien qu'au niveau des spécifications fonctionnelles, on peut appliquer la relation FAIT PARTIE DE aux éléments Fonction - Phase - Application - Projet, on ne retrouve pas cette relation et cette découpe au niveau des machines-phase et autres modules informatiques développés dans l'architecture d'une application informatique.

Si on examine plus particulièrement les critères de décomposition en modules, on retrouve quelques caractéristiques communes aux concepts de machine-phase et de module (que l'on définit le plus souvent par une unité de traitement compilable séparément) :

- Un module associé à une structure de données abstraites est également décrit par les propriétés de sa structure de données (choix du mode de représentation) et par les opérations exécutées sur ces données (algorithmes).
- Tout module ou composant d'une structure modulaire jouit de 3 qualités qui sont retrouvées dans toute machine-phase :
  * Forte capacité de cacher de l'information (c'est le cas pour la machine-phase puisque les personnes extérieures ne doivent pratiquement rien connaître de celle-ci)
  * Faible degré de couplage (c'est aussi le cas car on observe un très faible degré d'interdépendance entre machines-phase)
  * Forte cohésion interne (la sémantique et la logique de structure présentes dans une phase se retrouve en effet au niveau de la machine-phase)

==> En résumé, on peut affirmer que les concepts de module et de machine-phase possèdent donc de nombreux points communs mais ne sont toutefois pas exactement similaires.

Examinons à présent le concept de **machine abstraite.**

E.W. Dijkstra (dans [DIJKSTRA,68]) a démontré la valeur de la programmation utilisant des couches de machine abstraite (en tout cas pour la conception de systèmes d'exploitation). Cette hiérarchie de programmes est définie comme suit :

" Les parties du système sont des sous-programmes qui peuvent être appelés de la même façon que des procédures par un programme. Chaque sous-programme est supposé avoir un objectif bien défini.

La relation entre ces parties est la relation UTILISE définie comme suit:

UTILISE (pi, pj) ==== pi appelle pj ET
==== pj sera considéré incorrect si pj ne fonctionne pas correctement "

De plus, " Un programme divisé en un ensemble de sous-programmes est dit être structuré hiérarchiquement quand la relation UTILISE définit des niveaux comme décrit ci-après :

== Le niveau 0 est l'ensemble des parties pi de telle sorte qu'il n'existe pas de pj tel que UTILISE (pi,pj)
== Le niveau i est l'ensemble des parties pi tel que
a) il existe un pj au niveau i-1 tel que UTILISE (pi,pj)
b) si UTILISE (pi,pk) alors pk est au niveau i-1 ou plus bas " [PARNAS,72a]

Le terme "machine abstraite" est alors utilisé pour désigner ce programme décomposable et plus spécifiquement les composants des niveaux les plus hauts (>> 0) car la relation entre les programmes situés à un bas niveau et les programmes situés à un haut niveau est équivalente à la relation entre le hardware et le software.

EXEMPLE : Construction d'un Operating System par niveaux d'abstraction
Niveau 0 : Spécifications Hardware
.....
Niveau N : Spécifications Software, plus abstraites

Trois caractéristiques sont également à noter à ce sujet:

" - Le programme appelant doit être capable d'ignorer les traitements internes du programme appelé et le programme appelé ne doit faire aucune supposition à propos de la structure interne du programme appelant.
- Quand un programme a cette hiérarchie, les sous-programmes de bas niveau peuvent toujours être utilisés sans les sous-programmes de haut niveau quand ces derniers ne sont pas prêts ou que leurs services ne sont pas nécessités.
- Cette division de programmes en niveaux par la relation UTILISE n'a pas de connexion nécessaire avec la division des programmes en modules . " [PARNAS,72a]

La relation définie entre les modules informatiques est quasi-identique à la relation UTILISE définie par Dijkstra mais à ce stade-ci la structuration hiérarchique n'existe pas comme telle entre les modules informatiques (nous verrons néanmoins dans le point III.3 qu'une telle découpe en niveaux peut plus ou moins se retrouver pour les modules informatiques).

Chaque module informatique (et donc machine-phase) s'identifie aux composants reliés par cette relation UTILISE car ces composants s'appellent comme des procédures, ils sont considérés incorrects si le composant appelé ne fonctionne pas et chaque programme appelant ignore les traitements internes du programme appelé et vice-versa.

La notion de niveaux d'abstraction se retrouvera dans une certaine mesure pour les machines-phase car l'on définira plusieurs versions de spécification de celles-ci, chacune de ces versions se rapprochant d'une machine physique. Mais cette "abstraction" se limite au niveau des spécifications et est difficilement comparable avec les relations Hardware-Software .

==> La notion de machine abstraite n'est pas vraiment similaire à celle de machine-phase, une machine-phase contient en fait bien plus de sémantique qu'une machine abstraite (forme, structure,...).

On peut aussi établir une brève comparaison avec le concept de type de données abstrait ("Abstract Data Types") tel que vu par Sommerville par exemple .

D'une façon générale, " un type de données abstrait est défini et spécifié par 3 composantes principales :

(1) un nom de type
(2) une spécification optionnelle du domaine de valeurs pour le type
(3) une spécification des opérations disponibles sur le type "

[SOMMERVILLE,89]

Donc un type de données abstrait possède tout comme une machine-phase (bien qu'exprimé d'une autre façon) un ensemble de données (une memoire) et des opérations (possédant également des arguments et résultats) possibles sur ces données.

" L'implémentation du type abstrait est distincte de sa spécification. En effet, l'implémentation inclut des détails de représentation du type et l'implémentation des opérations définies sur ce type. Si la partie spécification reste inchangée, la partie implémentation peut être modifiée, en changeant peut-être la représentation, sans requérir de changements aux autres parties du programme qui utilisent le type de données abstrait. " [SOMMERVILLE,89]

Nous allons voir que ce principe sera plus ou moins utilisé pour les machines-phase où la partie visible des spécifications restera inchangée tandis que la partie principale, cachée pour l'extérieur, pourra être modifiée afin de changer de représentation et obtenir l'implémentation finale.

II.5 Langage de spécification

Un point important reste à éclaircir : quel langage sera utilisé pour exprimer au mieux le corps des procédures de la machine-phase c'est à dire l'objectif et les règles de traitement des fonctions de la phase ?

Les autres composants de la machine-phase possèdent un formalisme bien précis (la mémoire : schéma E/A, modèle relationnel,...; l'entête des procédures et les paramètres : signature selon le langage cible choisi). La machine-phase pourra donc être utilisée correctement par d'autres personnes en se servant de sa partie visible (entête des procédures, actions dynamiques et paramètres) mais comment sera exprimée la partie cachée que constitue le corps des différentes procédures ?

II.5.1 *Problématique*

Les objectifs du langage de description qui sera choisi sont la compréhensibilité, la généralité et la communicabilité entre personnes concernées par la conception de la machine-phase et de l'application toute entière. L'objectif de spécifications fonctionnelles directement exécutables reste évidemment valable pour cette problématique.

Le problème des langages de description destinés à la conception des traitements fût étudié par de nombreux auteurs. Sommerville résume assez bien la situation. Il favorise dans un premier temps l'utilisation de langages de programmation de haut niveau (tels que ADA ou PASCAL) pour décrire le niveau le plus bas de conception d'un logiciel.

" De tels langages de description sont des langages possédant des structures de contrôle qui permettent au concepteur d'inscrire la structure initiale de la solution logicielle au problème de conception.

L'avantage évident d'un tel langage est que la structure décrite est exécutable si un compilateur de langage est disponible mais les désavantages sont nombreux :

- les langages de programmation doivent être compilables et donc ne peuvent pas être facilement étendus pour englober de nouveaux concepts
- les types de données, les structures de données et les opérations disponibles en tant que primitives dans les langages de programmation sont souvent de relativement bas niveau. Cela signifie que la représentation de quelques constructions de haut niveau intuitivement simples (par exemple, une séquence sans borne d'agrégats donnés) devient parfois très détaillée et embrouillée.
- étant donné que la réflexion du concepteur est contrainte par le langage qu'il utilise, plus le niveau des constructions disponibles est bas, plus les idées de conception sont sujettes à être influencées par les constructions du langage.

- si une implémentation initiale d'une conception, spécifiée à l'aide d'un langage de programmation, doit par la suite être réimplémentée, il sera difficile d'effectuer cette réimplémentation dans un langage de plus haut niveau que le langage de description de la conception ce qui peut poser des problèmes évidemment. " [SOMMERVILLE,89]

Donc, plutôt que d'utiliser un langage de programmation existant en tant que véhicule pour l'expression de la conception, une meilleure alternative serait d'utiliser un langage de description destiné à la documentation et à la communication de conception de logiciel.

De tels langages utilisent les structures de contrôle familières des langages de programmation de haut niveau pour spécifier le flux de contrôle mais offrent également au concepteur une flexibilité considérable dans la description des opérations en permettant notamment un raffinement progressif des traitements nécessaires.

En ce qui concerne les spécifications formelles, on remarque qu'actuellement ces techniques de spécification sont loin d'être largement utilisées dans le développement industriel de logiciels [SOMMERVILLE,89]. Leurs avantages principaux les plus cités sont :

- le développement de telles spécifications permet une meilleure compréhension des besoins du logiciel et de sa conception
- ces spécifications peuvent être traitées automatiquement
- il est possible de prouver qu'un programme est conforme à sa spécification
- il est possible d'analyser ces spécifications en utilisant des méthodes mathématiques
- un système de prototypage peut être disponible

Cependant, nous n'utiliserons pas les spécifications formelles pour les raisons suivantes :

- ces nouvelles techniques n'ont pas encore vraiment prouvé leurs avantages quant à l'économie de temps et d'efforts dans le développement de logiciels
- ces techniques sont très peu connues de la plupart des informaticiens et elles sont peu familières aux notations habituelles utilisées par ceux-ci pour les spécifications (la méthode IDA en ce qui nous concerne)
- beaucoup de domaines d'application sont encore difficilement spécifiables à l'aide de ces techniques

Le point de vue qui sera adopté va dans le sens de ce que propose Sommerville mais est plus complet: selon [BODART-PIGNEUR,89] , " les formes et langages appropriés pour décrire les traitements que l'on propose habituellement sont : la langue naturelle, les règles de production, les tables de décision, les pseudo-codes, les langages algébriques et la logique des prédicats du premier ordre. Une approche déclarative est cependant souvent

préférée à une approche algorithmique étant donné que l'objectif de généralité des spécifications décrites est mieux respecté. En effet, dans l'approche déclarative l'ordre d'expression des règles n'a pas de signification et plus d'adaptations et remaniements sont possibles; par contre dans l'approche algorithmique, l'ordre d'exécution des règles est indiqué et la spécification devient immédiatemment solution (ce qui est un peu prématuré dans la démarche de conception).

Sous l'angle de la communication, l'emploi de la langue naturelle est recommandé ainsi que l'usage des règles de production et/ou de tables de décision combinatoires. "

Tenant compte de toutes ces remarques et pour d'autres raisons expliquées plus loin, il fût décidé de créer pour nos besoins précis un nouveau langage de spécification des traitements ou plus exactement d'étendre les notions d'un langage déjà existant.

II.5.2 *Langage DESPATH+*

Ce langage existant, c'est **DESPATH**, un langage de manipulation de bases de données représentées sous forme de modèles E/A.

" Ce langage utilisant la richesse sémantique des schémas E/A est purement descriptif, spécialement en ce qui concerne la partie Spécification de DESPATH (l'autre partie est la Modification) qui est orientée vers le langage naturel. Les spécifications en DESPATH peuvent être vues comme des chemins descriptifs à travers le schéma E/A qui sont exprimés via des clauses relatives chaînées et/ou logiquement connectées. " [ROESNER,85]

DESPATH constitue un langage simple, complet et de niveau assez haut qui permet aussi bien la manipulation de données (modification et consultation) que la spécification de vues de données. Pour examiner la syntaxe et la sémantique détaillées de DESPATH, il est conseillé de consulter le document [ROESNER,85] .

Le problème qui se pose avec le langage DESPATH est que les concepts de ce langage ne suffisent malheureusement pas à exprimer tous les traitements que l'on est susceptible de retrouver à cette étape de la conception. Dès lors, des modifications ont été apportées à DESPATH afin de pouvoir exprimer toutes les spécifications possibles, ce qui a fourni le langage de spécification semi-naturel (naturel structuré) **DESPATH+**.

Ces modifications consistent en une reformulation et une extension du langage original DESPATH. La plupart des instructions de DESPATH+ sont en fait une simple traduction en français accompagnée d'un légér remaniement des instructions DESPATH, auxquelles on a ajouté de nouvelles notions.

DESPATH+ offre donc les possibilités de description suivantes :
- Manipulation de données sous forme d'entités, d'associations et d'attributs
- Utilisation de données sous forme de variables
- Utilisation de primitives de désignation de données
- Utilisation de primitives de manipulation de données
- Utilisation de structures de contrôle

Exposons assez brièvement et d'une façon relativement simpliste les concepts principaux de ce langage de spécification . Nous présenterons ce langage selon 3 étapes :

1 . Enumération et définition abrégée des primitives et mots - clés
2. Explication détaillée et illustrée des primitives
3. Exposé des structures de contrôle

II.5.2.1. Enumération et définition abrégée des primitives et mots - clés

A . Verbes ou opérations

AJOUTER : permet l'insertion d'une (ou plusieurs) occurence(s) de type d'entité ou d'association

MODIFIER : permet la modification d'une (ou plusieurs) occurence(s) de type d'entité ou d'association et plus précisément des attributs spécifiés à la suite de MODIFIER.

SUPPRIMER : permet la suppression d'une (ou plusieurs) occurence(s) de type d'entité ou d'association

ASSIGNER A : permet l'assignation d'une valeur (de constante, d'attribut, d'entité ou association ou de variable) à une variable, attribut ou occurence(s) de type d'entité ou d'association

B . Mots clés (par ordre alphabétique)

AVEC :
permet la spécification des nouvelles valeurs d'attributs dans le cas des opérations AJOUTER et MODIFIER.

DE :
exprime le concept de projection d'un attribut sur une entité ou association, ou d'un composant de variable sur une variable.

DONT :
introduit la condition de sélection portant sur l'entité, l'association ou la variable considérée.

INFERIEUR (OU EGAL) A , SUPERIEUR (OU EGAL) A, EGAL A (PAS EGAL A) :
exprime la comparaison de valeurs (variables, attributs, entités, associations)
EGAL A est aussi utilisé pour l'assignation de valeurs dans le cas des opérations AJOUTER et MODIFIER.

ENTRE :
permet de spécifier les entités reliées par l'association considérée

ET / OU :
permet la conjonction de conditions ou de données

EXISTE UN (+ négation avec PAS)
EXISTE AU MOINS UN :
exprime le test de présence (ou d'absence) d'une valeur, variable, attribut , entité ou association

POUR UN ... (N') APPARTENANT (PAS) A (FIN POUR)
POUR PLUSIEURS ... (N') APPARTENANT (PAS) A :
règle le principe d'indétermination

QUI :
permet l'utilisation de noms de rôles afin de connecter plusieurs entités entre elles

SI ... SINON : exprime une condition dans la séquence d'instructions

TANT QUE .... FAIRE....FIN TANT QUE :
exprime la notion de boucle (utilisée dans les structures algorithmiques)

***NomdeProcédure* (*Liste de variables*) :**
appel à une procédure avec passage de paramètres

Ajoutons que les concepts mathématiques usuels MAX, MIN, COUNT, +, -,... font évidemment aussi partie des mots clés de ce langage et que certaines extensions peuvent être apportées pour régler des cas locaux (nouvelles opérations,...) .

II.5.2.2. Explication détaillée et illustrée des primitives

Nous décrirons les primitives du langage DESPATH+ en 2 parties : la première illustrera les primitives permettant de sélectionner des données et la deuxième énoncera les primitives destinées à modifier des données.

Examinons cela en signalant auparavant que :
- les mots entourés de [ ] forment une clause optionnelle à l'intérieur d'une construction
- les mots en majuscules sont des mots réservés
- les mots entourés de ( ) sont des clauses répetitives
- les mots séparés par des / dans des { } ne peuvent être utilisés simultanément mais au moins un des mots doit être utilisé.

A . Primitives de désignation de données (sélection) :

Conditions de sélection :

Il existe 2 types de base de propriétés possibles (qui peuvent être combinées bien sûr) à remplir par un sous-ensemble d'objets :
a) les valeurs d'attributs
b) la participation à des relations

a) { [<type d'attribut> DE] <type d'entité> / <type d'association> / <type de variable> }
[[DONT <type d'attribut>] {EGAL A / PAS EGAL A / SUPERIEUR A / INFERIEUR A} {<valeur> / <type de variable> / <type d'attribut> DE { <type d'entité> / <type d'association>} }]

b) [<type d'attribut> DE] <type d'entité> / <type d'association> QUI <nom de rôle> <type d'entité>

Nous pouvons donc sélectionner un sous-ensemble d'occurences de types d'attributs, d'entités, d'associations ou de variables respectant les conditions citées.
Il faut noter que ces constructions sont extensibles à souhait en respectant toujours cette syntaxe et que les clauses optionnelles peuvent être répétées en employant les mots ET / OU.

Exemple :
NUMCO DE COMMANDE DONT DATECO SUPERIEUR A DATEAPPR
DE PRODUIT QUI analogue PRODUIT DONT NUMP EGAL A 500

Nous aurons l'ensemble des numéros des commandes dont la date est supérieure à la date d'approvisionnement de tous les produits analogues au produit numéro 500.

Existence / Absence de valeurs :

{ EXISTE / EXISTE PAS } { UN / AU MOINS UN }
{ <type d'attribut> DE { <type d'entité> / <type d'association> } /
<type d'entité> / <type d'association> / <type de variable> }
[{ EGAL A / PAS EGAL A / SUPERIEUR A / INFERIEUR A }
{ <valeur> / <type d'attribut> DE { <type d'entité> / <type
d'association>} /
<type d'entité> / <type d'association> / <type de variable> }]

Cette construction sera essentiellement utilisée comme condition booléenne dans les structures conditionnelles SI .. SINON ou les structures itératives TANT QUE .
Les conditions de sélection vues auparavant sont bien sûr aussi utilisables dans les cas l'autorisant.

Exemple :
EXISTE UN N EGAL A NUMERO DE CLIENT DONT LOCALITE EGAL A "Genève"

Cette expression recherche si la variable N possède une valeur égale au numéro d'un des clients qui résident à Genève.

B . Primitives de manipulation de données

Nous trouverons 4 primitives de mise à jour des données à manipuler (base de données et variables).

Suppression :

SUPPRIMER <type d'entité>
[[DONT <type d'attribut>] { EGAL A / PAS EGAL A /
INFERIEUR A / SUPERIEUR A }
{ <valeur> / <type de variable> / <type d'attribut > DE
{ <type d'entité> / <type d'association> } }]

[QUI <nom de rôle> <type d'entité>]

On peut également supprimer un **<type d'association>** en utilisant la même syntaxe mais il faut rajouter en plus :

ENTRE <type d'entité> [Condition de sélection]
<type d'entité> [Condition de sélection]

Cette instruction permettra donc de supprimer les occurences de types d'entité ou d'association respectant éventuellement une (ou plusieurs) condition(s) de sélection .

Exemple :
SUPPRIMER LIGNE DONT QCOM EGAL A 10
ENTRE COMMANDE QUI Emise CLIENT DONT
NUMERO EGAL A 11
PRODUIT

Les associations LIGNE qui relie les entités PRODUIT et COMMANDE, dont la quantité commandée est égale à 10 et qui appartiennent aux commandes émises par le CLIENT numéro 11 seront supprimées.

Insertion :

AJOUTER { <type d'entité> / <type d'association> }
AVEC (<type d'attribut> EGAL A { <valeur> / <type d'attribut>
DE { <type d'entité> / <type d'association> } [Condition de sélection] / <type de variable> })
[ENTRE <type d'entité> [Condition de sélection]
<type d'entité> [Condition de sélection]]

Cette opération permet l'insertion d'occurence(s) de types d'entité (ou de types d'association éventuellement sélectionnés selon une ou plusieurs conditions portant sur les entités reliées).

Exemple :
AJOUTER CLIENT AVEC NOM EGAL A N
LOCALITE EGAL A "Namur"

Modification :

MODIFIER { <type d'entité> / <type d'association> }
[[DONT <type d'attribut>] { EGAL A / PAS EGAL A /
INFERIEUR A / SUPERIEUR A } { <valeur> /
<type de variable> / <type d'attribut > DE
{ <type d'entité> / <type d'association> } }]

[QUI <nom de rôle> <type d'entité>]

AVEC (<type d'attribut> EGAL A { <valeur> / <type d'attribut>
DE { <type d'entité> / <type d'association> } / <type de
variable> })

[ENTRE <type d'entité> [Condition de sélection]
<type d'entité> [Condition de sélection]]

Cette opération combine en fait les 2 précédentes : elle permet une sélection classique des types d'entité ou d'association à traiter (comme dans SUPPRIMER) et elle définit ensuite les nouvelles valeurs des types d'attribut à mettre à jour (comme dans AJOUTER à la seule différence qu'ici il ne faut pas spécifier tous les attributs mais seulement ceux à modifier).

Exemple :
MODIFIER CLIENT DONT NUMERO SUPERIEUR A 1000
AVEC NOM EGAL A NOM DE CLIENT
DONT NUMERO EGAL A 500

Les clients possédant un numéro > 1000 prendront comme valeur de nom le nom du client numéro 500.

Assignation :

ASSIGNER { <valeur> / <type de variable> / <type d'attribut> DE
{ <type d'entité> / <type d'association> } / <type
d'entité> / <type d'association> }
A { <type de variable> / <type d'attribut> DE {<type d'entité>
/ <type d'association> } / <type d'entité> / <type d'association>}
[Condition de sélection]

Les types des 2 objets impliqués dans l'opération doivent bien sûr être semblables ou compatibles (attention aux sous-ensembles d'objets sélectionnés).

Exemple :
ASSIGNER LOCALITE DE CLIENT DONT NUMERO ECAL A 100 A
LOCALITE DE CLIENT DONT CATEGORIE EGAL A "Douteux"

II.5.2.3. Exposé des structures de contrôle

Indétermination :

{ POUR UN / POUR PLUSIEURS } <type de variable>
{ APPARTENANT A / N'APPARTENANT PAS A }
{ <type d'attribut > DE { <type d'entité> / <type d'association> } /
<type d'entité> / <type d'association> / <type de variable> }
FIN POUR

La variable devra bien sûr être de même type que celui de l'objet spécifié après APPARTENANT A / N'APPARTENANT PAS A.
Les conditions de sélection sont toujours applicables selon les cas autorisés précédemment.

Condition :

SI { <expression basée sur EXISTE> / <Condition de sélection a> /
<Condition de sélection b> }
instructions
[SINON instructions]

L'utilisation des ET / OU est évidemment possible afin de considérer plusieurs clauses et les SI emboîtés sont également applicables.

Itération :

TANT QUE { <expression basée sur EXISTE> / <Condition de sélection a> /
<Condition de sélection b> }
FAIRE instructions
FIN TANT QUE

II.6 Illustration : Machine-phase GestionClients

Donnons à présent un exemple complet de machine-phase en spécifiant la machine correspondant à la phase Enregistrement d'un Client d'une application s'occupant de la gestion d'une firme de vente par correspondance (gestion des commandes, du stock,...).

Décrivons d'abord brièvement les spécifications fonctionnelles de la phase traitée. Elles sont décrites ici en français mais il ne faut pas oublier qu'elles seront traduites la plupart du temps en DSL.

La **phase Enregistrement d'un client** :
a pour **objectif** de vérifier si un client à enregistrer est valide, et lorsqu'il l'est, de le mémoriser. De plus, elle doit permettre de supprimer un client, d'enregistrer un changement de localité et de lister les clients d'une certaine catégorie.
utilise et/ou modifie une mémoire qui comprend des clients.
reçoit un client à enregistrer qui comprend : - un nom de client
- une localité de client
génère un client valide si la mémorisation a été possible.

Le sous-schéma E/A se limite à une entité **Client** caractérisée par :
- un numéro
- un nom
- une localité
- une catégorie
(avec comme valeurs : Régulier, Nouveau, Ancien ou Douteux)
et identifiée par son numéro.

La phase se compose de l'ensemble de fonctions suivantes :
- Fonction **MémorisationClient**
- Fonction **ValidationClient**
- Fonction **SuppressionClient**
- Fonction **ModificationLocalitéClient**
- Fonction **ListeClientsparCategorie**

Fonction MémorisationClient :
a pour **objectif** de créer et mémoriser un nouveau client
reçoit un Nom de client, une Localité de client
génère un Client valide
Ajoute un Client (dont le numéro est attribué par compostage,
le nom est celui donné en entrée,
la localité est celle donnée en entrée,
la limite d'achat est de FB 2500,
la catégorie est "Nouveau")

Fonction ValidationClient :
a pour **objectif** d'identifier un client, existant dans la mémoire, à partir de son seul numéro de client
reçoit un Numéro de client
génère un Client valide
(s'il existe dans la mémoire un client avec un numéro de client donné)

Fonction SuppressionClient :
a pour **objectif** de supprimer un client existant dans la mémoire
reçoit un Client valide
Supprime un Client (qui est égal à Client valide)

Fonction ModificationLocalitéClient :
a pour **objectif** d'enregistrer un changement de localité communiqué par un client qui existe déjà dans la mémoire
reçoit un Client valide et une Localité (nouvelle) de client
Modifie la localité du Client (en lui substituant la localité du nouveau domicile fourni)

Fonction ListeClientsparCategorie :
a pour **objectif** de sélectionner les clients, existant dans la mémoire, qui possèdent la valeur de catégorie donnée en entrée
reçoit une Catégorie de client
génère des Clients sélectionnés
Consulte Client

Le langage cible choisi pour le processus de dérivation des spécifications est le langage COBOL mais cela a peu d'importance ici. Le chapitre IV présentera en détail les spécificités de l'application des différents concepts dans l'environnement COBOL.

Les 4 composants de la machine-phase GestionClients qui correspond à cette phase sont définis comme suit :

II.6.1 *Mémoire*

La mémoire représentera donc un sous-schéma du schéma de données de l'application. La forme choisie ici fut le modèle Entité / Association mais un autre modèle aurait pu être envisagé aussi comme modèle de représentation (avec une adaptation conséquente du langage de description si nécessaire).

Expliquons brièvement les composants décrivant cette mémoire:

- *Entité (ou association) de base* explicite l'utilisation fixe définie sur l'E/A (entité ou association) principale que l'on décrit c'est à dire la dénomination selon laquelle sera utilisée l'E/A dans les traitements.

- *Entités (ou associations) temporaires* explicite les utilisations temporaires (particulières à certaines opérations) définies sur l'E/A principale que l'on décrit.

- *Types* permet de donner une spécification précise du type des données présentes en mémoire selon le formalisme de l'environnement cible (COBOL dans ce cas-ci).

- *Identifiant* permet de spécifier le (ou les) identifiant(s) de chaque E/A.

- *Domaines* spécifie l'ensemble des valeurs admises pour une donnée.
  Quand aucune valeur n'est spécifiée après le signe =, cela signifie que toute valeur vérifiant le type créé hormis la valeur vide, nulle ou inconnue peut être attribuée à cette donnée.
  Quand l'expression "... U VIDE" est spécifiée après le signe =, cela signifie que toute valeur vérifiant le type créé (y compris la valeur vide) peut être attribuée à cette donnée.
  Quand toute autre valeur est spécifiée après le signe =, cela signifie que les valeurs admises par la donnée se limitent à celles spécifiées.

- Les *contraintes d'intégrité* sont formulées selon le langage DESPATH+.

- Remarque quant aux variables locales de chaque procédure :
  Les variables locales définies pour les traitements de chaque procédure constituent en fait aussi des données de la mémoire de la machine-phase mais pour des raisons de facilité, celles-ci sont décrites ici après chaque entête de procédure.

Entité :
CLIENT : NUMERO
NOM
LOCALITE
CATEGORIE

Entité de base :
Clients(CLIENT)

Entités temporaires :
Clientsselectionnes(CLIENT)

Types (COBOL-LIKE) :

NUMERO	PIC 9(5)
NOM	PIC X(30)
LOCALITE	PIC X(30)
CATEGORIE	PIC X

Identifiant:
NUMERO

Domaines :
Dom(NUMERO) =
Dom(NOM) =
Dom(LOCALITE) =
Dom(CATEGORIE) = (R,N,D,C)

Contraintes d'intégrité :
1. AJOUTER CLIENT AVEC ..., CATEGORIE EGAL A N

II.6.2. *Paramètres*

Définissons à présent les paramètres tels qu'ils seront utilisés dans le futur programme COBOL et tels qu'ils seront appelés par les autres composants de l'application.

Parmi les règles qui seront appliquées afin de spécifier les paramètres des procédures selon une définition purement COBOL , citons celles-ci :

- Quand un paramètre est commun à plusieurs procédures de la machine-phase, il ne sera défini qu'une seule fois, les restrictions COBOL l'admettant (principe des ENTRY) et le principe de la connaissance extérieure du format et contenu des paramètres étant sauf.
- On utilisera 2 préfixes pour distinguer les paramètres des procédures :
  le préfixe L-I-... (INPUT) signale que l'on a affaire à un argument et le préfixe L-O-... (OUTPUT) signale que le paramètre est un résultat

Ces variables représentent donc les arguments et résultats de chaque procédure *à la sauce COBOL* . De cette façon, toute personne saura exactement comment utiliser une procédure en échangeant les données adéquates.

On a donc comme liste des paramètres (nom et type) des procédures de la machine-phase :

```
01 L-I-NOMC      PIC X(30).        (Nom du client)
01 L-I-LOCC      PIC X(30).        (Localité du client)
01 L-I-NUMC      PIC 9(5).         (Numéro du client)
01 L-I-CATC      PIC 9(5).         (Catégorie du client)
01 L-I-CLIVAL.                     (Client valide)
 05 L-I-CLIENT.
   10 L-I-NUMEROVAL     PIC 9(5)
   10 L-I-NOMVAL        PIC X(30)
   10 L-I-LOCALITEVAL   PIC X(30)
   10 L-I-CATEGORIEVAL  PIC X
```

```
01 L-O-CLIVAL.                    (Client valide)
  05 L-O-CLIENT.
    10 L-O-NUMEROVAL      PIC 9(5)
    10 L-O-NOMVAL         PIC X(30)
    10 L-O-LOCALITEVAL    PIC X(30)
    10 L-O-CATEGORIEVAL   PIC X

01 L-O-CLISEL.                    (Clients sélectionnés)
   05 L-O-TABCLI OCCURS 50 TIMES
      10 L-O-CLIENT.
         15 L-O-NUMEROVAL      PIC 9(5)
         15 L-O-NOMVAL         PIC X(30)
         15 L-O-LOCALITEVAL    PIC X(30)
         15 L-O-CATEGORIEVAL   PIC X

01 L-O-OK      PIC X.             (Résultat de test booléen)
```

II.6.3. *Procédures*

Chaque procédure correspondant à une fonction de la phase possède la signature COBOL suivante :

ENTRY "Nom de la procédure" USING "Nom des paramètres"

Procedure MEMORISATIONCLIENT :
ENTRY "MEMORCLI" USING L-I-NOMC L-I-LOCC
L-O-CLIVAL.

```
POUR UN N N'APPARTENANT PAS A NUMERO DE Clients
 AJOUTER Clients AVEC  NUMERO EGAL A N,
                       NOM EGAL A L-I-NOMC,
                       LOCALITE EGAL A  L-I-LOCC,
                       CATEGORIE EGAL A  "N"
FIN POUR
ASSIGNER N A  L-O-NUMEROVAL
ASSIGNER NOM A  L-O-NOMVAL
ASSIGNER LOCALITE A  L-O-LOCALITEVAL
ASSIGNER "N" A  L-O-CATEGORIEVAL
```

Procedure VALIDATIONCLIENT :
ENTRY "VALIDCLI" USING L-I-NUMC L-O-CLIVAL.

```
SI EXISTE UN NUMERO DE Clients EGAL A L-I-NUMC
          ASSIGNER  Clients  A L-O-CLIVAL
SINON     ASSIGNER  NUL    A L-O-CLIVAL
```

Procedure SUPPRESSIONCLIENT:
ENTRY "SUPPRCLI" USING L-I-CLIVAL.

Pré : EXISTE UN Clients EGAL A L-I-CLIVAL

SUPPRIMER Clients EGAL A L-I-CLIVAL

Procedure MODIFICATIONLOCALITECLIENT:
ENTRY "MODIFCLI" USING L-I-CLIVAL L-I-LOCC.

Pré : EXISTE UN Clients EGAL A L-I-CLIVAL

MODIFIER Clients EGAL A L-I-CLIVAL
AVEC LOCALITE EGAL A L-I-LOCC

Procedure LISTECLIENTSPARCATEGORIE
ENTRY "LISTCLIC" USING L-I-CATC L-O-CLISEL.

ASSIGNER Clients DONT CATEGORIE EGAL A L-I-CATC
A L-O-CLISEL

Nous voyons donc que la transformation du schéma de données, des fonctions, des messages et des règles de traitement peut s'opérer quasi-automatiquement et peut être régie par des règles précises (dénomination, valeurs par défaut,...) que nous ne détaillerons pas dans ce travail.

II.6.4. *Actions dynamiques*

Procedure INSTALLER :
ENTRY "INSTMACH" USING L-O-OK.

OUVRIR BD
SI BD OUVERTE ASSIGNER "V" A L-O-OK
SINON ASSIGNER "F" A L-O-OK

Procedure DESINSTALLER :
ENTRY "DESINMAC" USING L-O-OK.

FERMER BD
SI BD FERMEE ASSIGNER "V" A L-O-OK
SINON ASSIGNER "F" A L-O-OK

Les actions dynamiques (installation/désinstallation de la machine-phase) consistent uniquement à ouvrir ou fermer la base de données mise à disposition des procédures de la machine-phase.

Le principe de construction de la machine-phase semble en fait respecter un compromis entre deux perspectives de développement d'applications informatiques que l'on rencontre très souvent : la première est une **perspective industrielle** qui veut que l'on soit très technique, que l'on raisonne en termes de performances, de niveau physique (c'est ce que l'on fait en exprimant selon le langage cible la signature et les paramètres des procédures); la deuxième est une **perspective pédagogique** qui insiste sur la communicabilité et la compréhensibilité du travail (cela est pleinement assuré par le langage DESPATH+).

Ces deux pôles distincts sont donc bien nuancés au sein de cette machine-phase, nouvelle expression d'une phase IDA.

CHAPITRE III - TECHNIQUES DE TRANSFORMATION

Ce chapitre est destiné à présenter les différentes techniques qui pourront être appliquées aux composants de la machine-phase afin de raffiner les spécifications de celle-ci et arriver au but final qui est la production d'une machine physique, partie de l'application informatique finale (solution exploitable).

Le but général de ces différentes techniques est donc, sur la base du modèle général d'une machine-phase (et du modèle de base représenté par une phase), d'aboutir à un système exécutable par **dérivation continue** (c'est à dire sans modification ni perte de sémantique et sans changement du modèle de base) et par **dérivation systématique** (en appliquant des transformations bien définies aux éléments du modèle général) .

Les deux catégories principales de techniques qui sont proposées, bien que privilégiant toujours la continuité de la dérivation, favorisent surtout la souplesse d'utilisation des différentes transformations possibles. Elles offriront en effet au programmeur de multiples possibilités d'application des différentes techniques de transformation des composants de la machine-phase afin de déterminer quels sont les meilleurs choix de développement à faire.

L'application de ces différentes techniques sera coordonnée au sein de la démarche de dérivation proposée au chapitre V.

III.1 Transformation de signature [PIGNEUR,90]

La première catégorie de techniques de transformation s'applique à la signature des procédures des machines-phase et aux paramètres associés (correspondant aux messages reçus et générés) à ces procédures, c'est à dire la partie visible de la machine-phase.

Ces techniques ont comme but principal de simplifier les échanges d'informations entre la machine-phase et les autres modules informatiques définis. Les structures de données echangées sont en effet considérées comme trop complexes et trop riches (alors que peu d'informations sont réllement nécessaires aux procédures) et le système actuel exige des recopies fréquentes entre les données échangées, recopies qui s'avèrent souvent inutiles. De nouvelles alternatives seront proposées pour pallier à cette modification des échanges de données.

III.1.1 *Simplification de paramètres*

La technique de simplification de paramètres va permettre, comme on l'a dit, de simplifier les échanges de données entre modules informatiques, d'éviter les recopies fréquentes de données échangées mais aussi d'accéder à la seule information utile à la procédure et de faciliter la maintenance en cas de changements apportés aux structures de données.

Ce mécanisme consiste à remplacer, dans une procédure, chaque paramètre décomposable par un paramètre atomique (qui est identifiant) et à créer de nouvelles procédures de consultation pour les paramètres atomiques (non-identifiants) qui sont effectivement utilisés dans la procédure concernée.

La règle de simplification qui est d'usage s'exprime comme suit :
Les paramètres correspondent :
- soit à des structures de données de type E/A (structures présentes dans le sous-schéma de données de la phase et donc dans la mémoire de la machine-phase) purement et simplement et il suffira dès lors de remplacer le paramètre par sa référence identifiante à savoir l'identifiant de l'E/A correspondante.

- soit à des structures de données qui sont presque similaires à des E/A (incluant celles-ci) présentes dans le sous-schéma de données, on parlera alors d'E/A analogue. L'E/A "générique" (sur laquelle se base le paramètre) est alors simplifiée à son identifiant.

- soit à d'autres données ou structures de données: dans ce cas on ne représentera pas le paramètre par un identifiant car cela exigerait la création d'un nombre trop important de nouveaux identifiants et une gestion trop lourde des paramètres remplacés.

On ne simplifiera donc que les structures de données similaires aux structures présentes en mémoire de la machine-phase (la valeur de ces données étant présente dans la mémoire) car les désavantages engendrés par la simplification d'autres structures de données ne compenseraient pas les quelques avantages obtenus par cette simplification et seraient trop importants:
- Nécessité de création de nouvelles structures de données en mémoire centrale et gestion de celles-ci
- Foisonnement de procédures de consultation et d'appels à celles-ci
- Perte d'identité des paramètres par rapport aux messages initiaux de la phase

Pour chaque paramètre, la règle de simplification stipule 2 étapes :
1. Définir les composants non-simples du paramètre (décomposables . et/ou répétitifs)
2. Remplacer quand cela est possible les composants par leur identifiant (les composants remplaçables sont de type E/A ou analogue à un type E/A).

Illustrons cette technique en se servant de la machine-phase GestionClients qui a été explicitée au chapitre II.

Les **paramètres** L-I-CLIVAL et L-O-CLIVAL représentant des structures de données de type entité CLIENT qui ont été validées pourront être remplacés par leur **identifiant** à savoir L-I-NUMEROVAL dans les procédures qui les utilisent (MémorisationClient, ValidationClient, SuppressionClient et ModificationLocalitéClient).

Par exemple, le paramètre L-O-CLIVAL :

```
01 L-O-CLIVAL.                (Client valide)
 05 L-O-CLIENT.
   10 L-O-NUMEROVAL      PIC 9(5)
   10 L-O-NOMVAL         PIC X(30)
   10 L-O-LOCALITEVAL    PIC X(30)
   10 L-O-CATEGORIEVAL   PIC X
```

devient :

```
01 L-O-CLIVAL.                (Client valide)
 05 L-O-CLIENT.
   10 L-O-NUMEROVAL      PIC 9(5)
```

et la procédure MémorisationClient devient :

```
Procedure MEMORISATIONCLIENT :
ENTRY "MEMORCLI" USING L-I-NOMC  L-I-LOCC
                       L-O-CLIVAL.

POUR UN N N'APPARTENANT PAS A NUMERO DE Clients
 AJOUTER Clients AVEC  NUMERO EGAL A N,
                       NOM EGAL A L-I-NOMC,
                      LOCALITE EGAL A  L-I-LOCC,
                       CATEGORIE EGAL A  "N"
FIN POUR
ASSIGNER N A  L-O-NUMEROVAL
```

Il faudra également créer de nouvelles procédures qui consulteront la mémoire sur base de l'identifiant L-O-NUMEROVAL (que l'on transférera dans l'argument L-I-NUMC des procédures décrites ci-dessous) et rendront comme résultats les paramètres atomiques non-identifiant de L-I-CLIVAL et L-O-CLIVAL qui sont utilisés dans les procédures concernées (c'est à dire L-O-NOMC, L-O-LOCC et L-O-CATC dans les 3 procédures de consultation décrites ci-dessous).

Ces procédures posséderont toujours comme seul argument l'identifiant du paramètre remplacé et comme seul résultat un paramètre atomique non-identifiant, composant du paramètre initial remplacé, cela principalement afin de permettre la réutilisabilité des procédures créées.

Signalons aussi qu'afin d'éviter de créer 2 fois une même procédure de consultation (lorsqu'elle a déjà été créée suite à la traduction d'une autre fonction), on disposera d'un tableau "marqué" qui indiquera quelles procédures doivent être créées pour les besoins de la fonction.

Dans le cas de L-I-CLIVAL ou L-O-CLIVAL, nous avons 3 paramètres atomiques (Nom, Localité et Catégorie) et nous aurons donc au maximum les 3 procédures de consultation suivantes :

Procedure NOMDECLIENT :
ENTRY "NOMDECLI" USING L-I-NUMC L-O-NOMC.

Pré : EXISTE UN NUMERO DE Clients EGAL A L-I-NUMC

ASSIGNER NOM DE Clients DONT NUMERO EGAL A L-I-NUMC
A L-O-NOMC

Procedure LOCALITEDECLIENT :
ENTRY "LOCDECLI" USING L-I-NUMC L-O-LOCC.

Pré : EXISTE UN NUMERO DE Clients EGAL A L-I-NUMC

ASSIGNER LOCALITE DE Clients DONT NUMERO EGAL A L-I-NUMC
A L-O-LOCC

Procedure CATEGORIEDECLIENT :
ENTRY "CATDECLI" USING L-I-NUMC L-O-CATC.

Pré : EXISTE UN NUMERO DE Clients EGAL A L-I-NUMC

ASSIGNER CATEGORIE DE Clients DONT NUMERO EGAL A L-I-NUMC
A L-O-CATC

Les procédures MémorisationClient, ValidationClient, SuppressionClient et ModificationLocalitéClient feront alors appel à ces procédures si leurs traitements utilisent effectivement les paramètres atomiques non-identifiant nom, localité ou catégorie.

Mais ce mécanisme de simplification aura des conséquences sur d'autres composants de la machine-phase :

- la correspondance *Fonction d'une phase - Procédure d'une machine-phase* ne sera en effet plus biunivoque (1-1) car le fait de créer ou d'utiliser de nouvelles procédures de consultation donnera une correspondance Fonction-Procédure 1-N avec N>=1.

- l'application du mécanisme de simplification sur les arguments provoquera des conséquences quant à la formulation des préconditions (syntaxe) mais pas sur le contenu sémantique de celles-ci. Même si les arguments initiaux ont été remplacés par leur identifiant, les propriétés relatives à ces arguments initiaux restent toujours valables et devront encore être vérifiées au début de la procédure.

- les spécificités engendrées par la simplification des paramètres en ce qui concerne la spécification des traitements sont :
  * Pour les résultats qui ont été remplacés par leur identifiant : les traitements assurant leur génération se limiteront à une génération de l'identifiant mais les autres traitements opérant sur l'identifiant et les autres composants du résultat initial seront présents dans leur intégralité
  * Pour les arguments qui ont été remplacés par leur identifiant : il faudra faire appel aux nouvelles procédures de consultation à l'intérieur des traitements définis puisque tous les composants des arguments initiaux ne seront pas disponibles directement pour la procédure comme cela se passait auparavant. On remplacera simplement le nom du composant manquant par l'appel à sa procédure de consultation correspondante.

Le désavantage principal de ce mécanisme est qu'il faut définir de nouvelles procédures, ce qui représente un travail assez fastidieux et répétitif (mais facilement automatisable) et surtout le fait que les modules informatiques appelant une procédure de la machine-phase (traduisant une fonction de la phase) ne disposent pas immédiatemment de toutes les informations prévues par les spécifications fonctionnelles après l'exécution de cette procédure, ce qui est assez ennuyeux pour les modules utilisant les services de la machine-phase.

III.1.2 *Réduction de paramètres*

La technique de réduction de paramètres va permettre de simplifier les échanges de données et plus spécifiquement les générations de résultats par les procédures. Elle consiste à supprimer un résultat lorsque celui-ci est identique à un paramètre reçu c'est à dire un argument et de remplacer ce résultat par un simple booléen dans la majorité des cas. Cela permet de supprimer les échanges de données complètement inutiles.

Si cette technique est appliquée après une simplification de paramètres, il faudra vérifier que tous les autres composants des paramètres remplacés par leur identifiant n'ont pas été modifiés en valeur par les traitements afin de pouvoir procéder à la suppression du résultat considéré. Mais il est préférable et conseillé d'appliquer cette technique de réduction avant la simplification de paramètres afin de ne pas biaiser le mécanisme.

La règle d'usage s'exprime comme suit :
Tout résultat qui n'aura pas été modifié par les traitements de la procédure c'est à dire tout résultat strictement égal à un argument (de même type) et dont aucune composante n'a reçu de valeur nouvelle (à part la valeur NULL) par rapport à la valeur initiale de l'argument (à l'aide de l'instruction ASSIGNER ...A) sera supprimé et remplacé par un booléen quand l'objectif de la procédure l'exige.

Ce mécanisme aura des conséquences sur les traitements portant sur ces résultats réduits : tous les traitements assurant la génération des résultats réduits sont aussi éliminés. Dans certains cas de génération conditionnelle, il faudra ajouter une assignation à un résultat booléen indiquant la condition de sortie de la procédure.

Illustrons cela avec la procédure ValidationClient où le client valide généré (L-O-CLIVAL) est supprimé (car identique après simplification à L-I-NUMC) et remplacé par un booléen de même nom:

```
Procedure VALIDATIONCLIENT :
ENTRY "VALIDCLI" USING L-I-NUMC
                       L-O-CLIVAL.

SI EXISTE UN NUMERO DE Clients EGAL A L-I-NUMC
          ASSIGNER "V" A L-O-CLIVAL
SINON     ASSIGNER "F" A L-O-CLIVAL
```

III.1.3 *Répétition de résultats*

La technique de répétition de résultats a pour objectif de supprimer les résultats répétitifs générés par une procédure en transformant ceux-ci en une mémoire temporaire et en remplaçant la procédure génératrice par un mécanisme itérateur, sous la forme par exemple de 3 autres procédures effectuant les actions suivantes :

Initialisation de l'itération
Terminaison
Suivant

Illustrons ce mécanisme à l'aide de la procédure ListeClientsparCategorie qui génère un tableau de clients sélectionnés selon le critère de catégorie.

Le résultat répétitif L-O-CLISEL (ensemble de clients valides) est supprimé et de nouveaux paramètres sont introduits pour construire le mécanisme itérateur :

L-O-NUM PIC 9(5) représentant le numéro du client courant sélectionné
L-O-ITERM PIC X représentant le code de retour du test de terminaison

De plus, une relation temporaire sur le type d'entité CLIENT est définie : Clientsselectionnes (de même type que Clients), et elle constituera la mémoire temporaire remplaçant le résultat répétitif supprimé.

La procédure ListeClientsparCatégorie :

```
Procedure LISTECLIENTSPARCATEGORIE
ENTRY "LISTCLIC" USING L-I-CATC  L-O-CLISEL.

ASSIGNER Clients DONT  CATEGORIE EGAL A  L-I-CATC
                A  L-O-CLISEL
```

est remplacée par le mécanisme itérateur suivant :

```
Procedure INITITERATION :
ENTRY "INITITER" USING L-I-CATC.

ASSIGNER  Clients DONT CATEGORIE EGAL A L-I-CATC
                A  Clientsselectionnes


Procedure TERMITERATION :
ENTRY "TERMITER" USING L-O-ITERM.

SI EXISTE PAS AU MOINS UN Clientsselectionnes
                        ASSIGNER "V"  A L-O-ITERM
SINON                   ASSIGNER  "F"  A L-O-ITERM


Procedure SUIVITERATION :
ENTRY "SUIVITER" USING L-O-NUM.

Pré : EXISTE AU MOINS UN Clientsselectionnes

POUR UN C APPARTENANT A Clientsselectionnes
        SUPPRIMER Clientsselectionnes
        ASSIGNER NUMERO DE C  A  L-O-NUM
FIN POUR
```

Ce mécanisme permet également une simplification des structures de données échangées mais il introduit une mémoire temporaire qui exige une gestion supplémentaire à l'aide de nouvelles procédures. Cela présente l'inconvénient de devenir de plus en plus difficilement identifiable aux spécifications fonctionnelles de la fonction de départ.

Pour cette dernière raison, les avantages d'un tel mécanisme semblent moins évidents que pour les deux premières techniques.

III.1.4 *Répétition d'arguments*

La technique de répétition d'arguments consiste à supprimer les arguments répétitifs reçus par une procédure en transformant ceux-ci en une mémoire temporaire et en adaptant la procédure réceptrice. Cette adaptation se fera à l'aide de l'introduction d'un mécanisme accumulateur sous la forme, par exemple, de 3 procédures effectuant les actions suivantes :

Initialisation de l'accumulation
Insertion
Suppression

Illustrons ce mécanisme à l'aide d'une nouvelle procédure. La procédure ValidationCommande a pour objectif de valider une commande et reçoit comme arguments une ensemble de lignes de commande valide (L-I-LIGNESVAL) et un client valide (L-I-CLIVAL)

```
01 L-I-LIGNESVAL                          (Collection de lignes valides)
   05 L-I-LIGNEVAL  OCCURS 20 TIMES
      10 L-I-PRODUIT.                          (Produit)
         15 L-I-NUMPR        PIC 9(5)
         15 L-I-LIBELLE      PIC X(30)
         15 L-I-UNITACH      PIC 9(3)
         15 L-I-PRIXUNIT     PIC 9(5)
         15 L-I-ETATAPPR     PIC X
         15 L-I-DATEAPPR     PIC 9(6)
         15 L-I-PRODAN       PIC 9(5)
      10 L-I-QCOMVAL         PIC 9(3)      (Quantité commandée)
            10 L-I-MONTANTLVAL  PIC 9(8)    (Montant total d'une ligne)

01 L-I-CLIVAL.                              (Client Valide)
   05 L-I-CLIENT.
      10 L-I-NUMEROVAL       PIC 9(5)
      10 L-I-NOMVAL          PIC X(30)
      10 L-I-LOCALITEVAL     PIC X(30)
      10 L-I-CATEGORIEVAL    PIC X
```

La procédure ValidationCommande :

Procedure VALIDATIONCOMMANDE :
ENTRY "VALIDCOM" USING L-I-LIGNESVAL
L-I-CLIVAL .....
.....

est remplacée par les 3 procédures assurant le mécanisme accumulateur:

Procedure VALIDATIONCOMMANDE :
ENTRY "VALIDCOM" USING L-I-CLIVAL .....
.....

Procedure INSERERLIGNE:
ENTRY "INSERLIG" USING L-I-LIGNEVAL
.....

Procedure SUPPRIMERLIGNE:
ENTRY "SUPPRLIG" USING L-I-LIGNEVAL
.....

De plus une mémoire temporaire contenant les lignes validées a été créée et est mise à jour par les procédures INSERERLIGNE et SUPPRIMERLIGNE. Cette mémoire temporaire remplace donc l'argument répétitif L-I-LIGNESVAL qui a été supprimé dans la procédure ValidationCommande. Cette dernière procédure et les procédures qui l'appeleront feront appel à ces procédures et consulteront la mémoire temporaire pour effectuer les traitements prévus.

Ce mécanisme offre des avantages et inconvénients semblables à ceux rencontrés pour la technique précédente.

En fait, ces deux dernières techniques sont des cas particuliers (limités aux paramètres répétitifs) d'une technique plus générale qui consiste à introduire et gérer une mémoire d'exécution (mémoire temporaire) au sein de la machine-phase, réduisant de la sorte les échanges de données avec les autres modules informatiques.

En effet, on remarque souvent que certaines données ne peuvent être mémorisées qu'ensemble (exemple : dans une phase Enregistrement d'une commande : le client et les lignes de commande ne seront mémorisées que simultanément avec la commande, après validation complète) alors que l'on connaît certaines de ces données avant et que l'on est presque certain de devoir les mémoriser à terme (exemple : Client avant Commande).

Il faut donc les mémoriser temporairement dans une mémoire à court terme en utilisant, dans beaucoup de cas, pour cette mémoire la même structure (ou une partie de celle-ci) que la mémoire à long terme de la machine-phase. La mémorisation temporaire d'une donnée s'effectuera en général après la validation de cette donnée par une procédure.

Cela permettra de simplifier encore davantage la structure et le nombre des données échangées entre procédures et modules (surtout en supprimant un grand nombre d'arguments), mais cela nécessitera cependant une transformation supplémentaire des traitements des procédures et une gestion assez complexe de la nouvelle mémoire créée (nouvelles procédures d'insertion, de suppression, de modification,...)

Cette technique semble en fait apporter plus d'inconvénients que de véritables avantages (les gains du point de vue des performances physiques de la machine n'étant pas l'objet d'études à ce stade de la conception).

III.2 Raffinement de la machine-phase

La deuxième catégorie de techniques de transformation s'applique à la partie cachée de la machine-phase à savoir la mémoire et le corps des procédures.

Ces techniques ont pour but de permettre au programmeur d'affiner, de détailler les spécifications techniques obtenues pour s'approcher de la solution exploitable. Elles lui offrent la possibilité d'essayer plusieurs architectures de traitements et de mémoire afin d'obtenir la meilleure possible.

La partie visible de l'extérieur reste donc inchangée ce qui permet au programmeur responsable du développement de la machine-phase de travailler indépendamment des autres concepteurs du projet.

Les trois premières transformations proposées sont indépendantes des solutions informatiques (physiques) potentielles et la quatrième transformation traduit la machine-phase dans le formalisme de l'environnement exécutable choisi.

III.2.1 *Changement de représentation*

La technique de changement de représentation va permettre de modifier la description de la mémoire de la machine-phase ce qui entraînera évidemment une adaptation correspondante de la spécification du corps des procédures.

Les raisons et caractéristiques classiques inhérentes à l'application de cette technique sont :

- ce type de changement permet au programmeur de découvrir de nouvelles perspectives, de modifier parfois complètement sa façon de considérer la machine-phase et plus tard le programme
- un changement de représentation de mémoire est essentiel et ses conséquences sur les autres parties de la machine-phase sont en effet importantes. La plupart des données manipulées dans la machine-phase ainsi que les traitements à effectuer dépendent du format de cette mémoire.
- afin de considérer plusieurs possibilités en fonction de l'environnement-cible et de son système de gestion de base de données, il faut approcher des représentations différentes de la mémoire utilisée

Illustrons cela à l'aide de la procédure MémorisationClients qui a été définie auparavant.

La mémoire est modifiée comme suit : l'entité originelle CLIENT est subdivisée en deux nouvelles entités et une association les reliant .

Entité :
CLIENT' : **NUMERO**
NOM
CATEGORIE

Entité de base :
Clients'(CLIENT')

Entités temporaires :
Clientsselectionnes'(CLIENT')

Types (COBOL-LIKE) :
NUMERO PIC 9(5)
NOM PIC X(30)
CATEGORIE PIC X

Identifiant :
NUMERO

Domaines :
Dom(NUMERO) =
Dom(NOM) =
Dom(IDLOCALITE) =
Dom(CATEGORIE) = (R,N,D,C)

Contraintes d'intégrité :
1. AJOUTER CLIENT AVEC ..., CATEGORIE EGAL A N

CLIENT' joue le rôle de CLIENT' (localité) dans une et une seule LOCALISATION

Entité:
LOCALITE : LOCALITE

Entité de base :
Localités(LOCALITE)

Types (COBOL-LIKE) :
LOCALITE PIC X(30)

Identifiant:
LOCALITE

Domaines :
Dom(LOCALITE) =

LOCALITE joue le rôle de LOCALITE (DeClient) dans une ou plusieurs LOCALISATION

Association:
LOCALISATION

Association de base :
Localisation (LOCALISATION)

Rôles :
LOCALITE (DeClient) : 1 - N
CLIENT' (localité) : 1 - 1

La procédure MémorisationClient voit ses traitements modifiés suite à ce changement:

```
Procedure MEMORISATIONCLIENT :
ENTRY "ENREGCLI" USING L-I-NOMC  L-I-LOCC  L-O-NUMC.

N  PIC 9(5)

POUR UN N N'APPARTENANT PAS A NUMERO DE Clients'
 SI EXISTE UN LOCALITE DE Localites EGAL A L-I-LOCC
                AJOUTER Clients' AVEC NUMERO EGAL A N,
                                      NOM EGAL A L-I-NOMC,
                                      CATEGORIE EGAL A  "N"
                AJOUTER Localisation
                     ENTRE Clients'  DONT NUMERO EGAL A N
                           Localités  DONT LOCALITE EGAL A L-I-LOCC
 SINON
                AJOUTER Clients'  AVEC NUMERO EGAL A N,
                                      NOM EGAL A L-I-NOMC,
                                      CATEGORIE EGAL A "N"
                AJOUTER Localités  AVEC  LOCALITE EGAL A L-I-LOCC
                 AJOUTER Localisation
                     ENTRE Clients'  DONT NUMERO EGAL A N
                           Localités DONT LOCALITE EGAL A L-I-LOCC

  ASSIGNER N A  L-O-NUMC
FIN POUR
```

Le changement de représentation de la mémoire a induit une modification importante des traitements à effectuer pour enregistrer un client puisque le simple ajout de celui-ci dans l'entité Clients est maintenant remplacé par un ajout dans l'entité Clients' et l'association Localisation dans le cas où la localité est déjà présente dans la base de données **ou** par un ajout dans les entités Clients' et Localités et l'association Localisation quand la nouvelle localité n'est pas encore présente. Evidemment des tests supplémentaires et des variables locales ont dû être ajoutés. La décomposition du type d'entité Clients en 2 types d'entité a donc induit 2 types d'ajout à traiter séparément.

Il est évident que les modifications les plus courantes de la mémoire seront des décompositions ou regroupements d'entités / associations, la préservation de la cohérence globale ne permettant bien sûr pas n'importe quelle fantaisie.

III.2.2 *Utilisation d'un module auxiliaire*

La technique d'utilisation d'un module auxiliaire va permettre d'introduire une nouvelle organisation (dans le sens de répartition) des traitements en créant et/ou utilisant des modules auxiliaires de même structure et forme qu'une machine-phase, et dont les procédures (vues comme des services) exécuteront une partie des traitements à effectuer par la machine-phase appelante.

Cette technique s'inscrit parfaitement dans la perspective de modularité et de hiérarchisation d'une application informatique.

On peut trouver deux types généraux de modules auxiliaires :
- les **modules auxiliaires techniques** que l'on peut également nommer modules de gestion de base de données. Ces modules sont chargés d'exécuter au sein de leurs procédures tous les traitements relatifs à une mémoire bien définie, partie ou totalité de la base de données de l'application, et d'assurer notamment tous les accès et opérations de mise à jour de celle-ci. On distingue souvent 2 grands types de traitements (et éventuellement de module auxiliaire technique) : les traitements de modification de la mémoire et les traitements de contrôle (validation) ou de consultation de la mémoire.

- les **modules auxiliaires utilitaires** (appelés aussi fonctionnels) offrant un des 2 types possibles de services aux modules le demandant :
  * les procédures communes ou génériques à plusieurs modules informatiques
  * les procédures fournissant des services divers regroupés logiquement au sein d'un module auxiliaire (services de conversion de format, services de recopie de données,....)

Les appels aux services de ces modules auxiliaires peuvent se faire directement à partir d'une machine-phase mais aussi à partir d'un autre module auxiliaire si cela s'avère nécessaire.

Les modules auxiliaires sont des modules informatiques à part entière puisqu'ils sont spécifiés d'une façon strictement identique aux machines-phase et que toutes les techniques de transformation évoquées dans ce chapitre leur sont également applicables.

Bien sûr la consolidation des traitements a été opérée au niveau des spécifications fonctionnelles, mais cette technique offre en plus la possibilité d'utiliser au mieux tout ce qui est déjà existant dans les autres modules et de réorganiser les spécifications techniques des traitements en vue d'obtenir une économie de spécification et de codage ainsi qu'un regroupement plus approfondi des idées de développement similaires ou proches.

Illustrons cette technique à l'aide de la nouvelle procédure MémorisationClient (dans sa version du point III.2.2) qui utilise à présent, pour remplir son objectif, les services de deux modules auxiliaires techniques.

Le premier, GestionMemoireClients, contient toutes les procédures de gestion de l'entité Clients' et de l'association Localisation; le deuxième, GestionMemoireLocalités, contient toutes les procédures de gestion de l'entité Localités.

La procédure MémorisationClient de la machine-phase se présente dès lors comme suit :

```
Procedure MEMORISATIONCLIENT ;
ENTRY "ENREGCLI" USING   L-I-NOMC  L-I-LOCC  L-O-NUMC.

PRESENT    PIC X
N          PIC 9(5)

VALIDLOC ( L-I-LOCC, PRESENT)
SI PRESENT = "T"   INSERCLI  (L-I-NOMC, L-I-LOCC, L-O-NUMC)
   SINON           INSERLOC (L-I-LOCC)
                   INSERCLI  (L-I-NOMC, L-I-LOCC, L-O-NUMC)
```

Une nouvelle variable locale a été définie (PRESENT, variable booléenne) et les traitements de la procédure se résument à présent à 4 appels de procédure et un test.

Le module auxiliaire GestionMemoireClients possède comme mémoire l'entité Clients' et l'association Localisation, et contient notamment les procédures suivantes où de nouveaux paramètres sont bien sûr définis:

```
Procedure INSERCLI :
ENTRY "INSERCLI" USING  L-BD-I-NOMC     L-BD-I-LOCC
                        L-BD-O-NUMC.
Objectif : Enregistrer un nouveau client dans l'entité CLIENT' et son association
           correspondante

POUR UN L-BD-O-NUMC N'APPARTENANT PAS A NUMERO DE Clients'
      AJOUTER Clients' AVEC  NUMERO EGAL A L-BD-O-NUMC,
                             NOM EGAL A L-BD-I-NOMC,
                             CATEGORIE EGAL A  "N"
      AJOUTER Localisation
                  ENTRE  Clients'  DONT NUMERO EGAL A L-BD-O-NUMC
                         Localités DONT LOCALITE EGAL A L-BD-I-LOCC
FIN POUR
```

Le module auxiliaire GestionMemoireLocalités possède comme mémoire l'entité Localités et contient notamment les procédures suivantes où de nouveaux paramètres sont bien sûr aussi définis:

Procedure VALIDLOC:
ENTRY "VALIDLOC" USING L-BD-I-LOCC L-BD-O-PRESENT.
Objectif : Tester la présence de la localité L-BD-I-LOCC dans l'entité Localités

SI EXISTE UN LOCALITE DE Localités EGAL A L-BD-I-LOCC
ASSIGNER "V" A L-BD-O-PRESENT
SINON ASSIGNER "F" A L-BD-O-PRESENT

Procedure INSERLOC:
ENTRY "INSERLOC" USING L-BD-I-LOCC.
Objectif : Insérer la localité de nom L-BD-I-LOCC dans l'entité Localités

Pré : EXISTE PAS UN LOCALITE DE Localités EGAL A L-BD-I-LOCC

AJOUTER Localités AVEC LOCALITE EGAL A L-BD-I-LOCC

.....

Les procédures de ces deux modules auxiliaires ne sont évidemment pas dérivées directement de fonctions figurant dans les spécifications fonctionnelles mais sont une création se situant au niveau des spécifications techniques. Ces procédures pourront être utilisées par tous les modules informatiques le désirant (la signature des fonctions et les paramètres étant visibles pour tous).

III.2.3 *Construction des algorithmes*

La technique de mise sous forme algorithmique va permettre de passer d'un mode de spécification des traitements purement déclaratif à un mode de description procédural.

L'objectif est de se rapprocher progressivement de la solution informatique sous forme exécutable étant donné que l'on suppose que la grande majorité des langages de programmation privilégient l'aspect procédural (bien que cette constatation semble s'infirmer au vu de l'évolution actuelle). Si l'environnement cible (et notamment le langage de programmation) favorise l'aspect déclaratif, il est évident que cette technique ne sera pas appliquée .

Cette technique introduit donc une nouvelle organisation des traitements au sein de chaque procédure en éliminant toute définition déclarative de ceux-ci et plus particulièrement en supprimant l'indéterminisme et le parallélisme, et en introduisant de nouvelles structures à savoir la boucle et la séquence.

Les transformations classiques peuvent être décrites comme suit :
- l'indétermination (POUR UN ... APPARTENANT A ...) est remplacée la plupart du temps par une itération (TANT QUE ...) ou une assignation précise de valeur (ASSIGNER A ...)

- l'élimination du parallélisme des traitements ne demande souvent pas beaucoup de modifications, mais juste une mise en commun de traitements définis plusieurs fois et un placement adéquat de ceux-ci dans la séquence d'exécution des opérations. Il faut bien sûr s'assurer à présent de l'ordre logique de tous les traitements du point de vue de leur exécution.

Illustrons cela une nouvelle fois à l'aide de la procédure MémorisationClient, reprise pour cet exemple dans sa version initiale (sans changement de représentation et sans utilisation de module auxiliaire) :

```
Procedure MEMORISATIONCLIENT :
ENTRY "MEMORCLI" USING  L-I-NOMC   L-I-LOCC   L-O-CLIVAL.

ASSIGNER (MAX(NUMERO DE Clients) + 1)  A  N
AJOUTER Clients AVEC    NUMERO EGAL A N,
                        NOM EGAL A L-I-NOMC,
                        LOCALITE EGAL A  L-I-LOCC,
                        CATEGORIE EGAL A  "N"
```

```
ASSIGNER N A L-O-NUMEROVAL
ASSIGNER NOM A L-O-NOMVAL
ASSIGNER LOCALITE A L-O-LOCALITEVAL
ASSIGNER "N" A L-O-CATEGORIEVAL
```

Remarquons que la transformation de l'indétermination en assignation de (MAX(NUMERO de Clients) + 1) à N est effectuée juste avant l'ajout dans Clients et les assignations aux composants de L-O-CLIVAL.

III.2.4 *Traduction en code exécutable (Programmation)*

La technique de traduction en code exécutable va permettre de traduire les spécifications techniques de la machine-phase dans le langage de programmation de l'environnement physique choisi.

La version finale de la machine-phase, c'est à dire une version sous forme exécutable, sera alors disponible, toutes les composantes de la machine-phase ayant été traduites correctement.

La description de la machine-phase ne peut évidemment plus être assimilée à des spécifications à ce stade-ci du développement mais purement et simplement à de la programmation.

Il est évident que cette technique sera généralement appliquée à la fin du processus de dérivation établi.

Une caractéristique importante de cette technique réside dans le fait que ce soit l'aspect syntaxique qui est étudié ici, alors que les autres techniques se préoccupaient davantage des questions sémantiques qui se posaient.

La procédure MémorisationClient deviendrait dans le cas du langage COBOL couplé au langage SQL-DS :

```
Procedure MEMORISATIONCLIENT :
ENTRY "MEMORCLI" USING  L-I-NOMC  L-I-LOCC  L-O-CLIVAL.

MOVE L-I-NOMC TO NOMC
MOVE L-I-LOCC TO L
MOVE "N" TO CATC
SELECT  MAX(NUMERO)+1
          INTO NUMC
          FROM CLIENT
INSERT INTO CLIENT (NUMERO,NOM,LOCALITE,CATEGORIE)
        VALUES (NUMC,NOMC,L,CATC)
```

```
MOVE NUMC TO L-O-NUMEROVAL
MOVE NOMC TO L-O-NOMVAL
MOVE L TO L-O-LOCALITEVAL
MOVE CATC TO L-O-CATEGORIEVAL
GOBACK
```

avec NUMC, NOMC, L et CATC comme variables servant d'interface entre COBOL et SQL-DS.

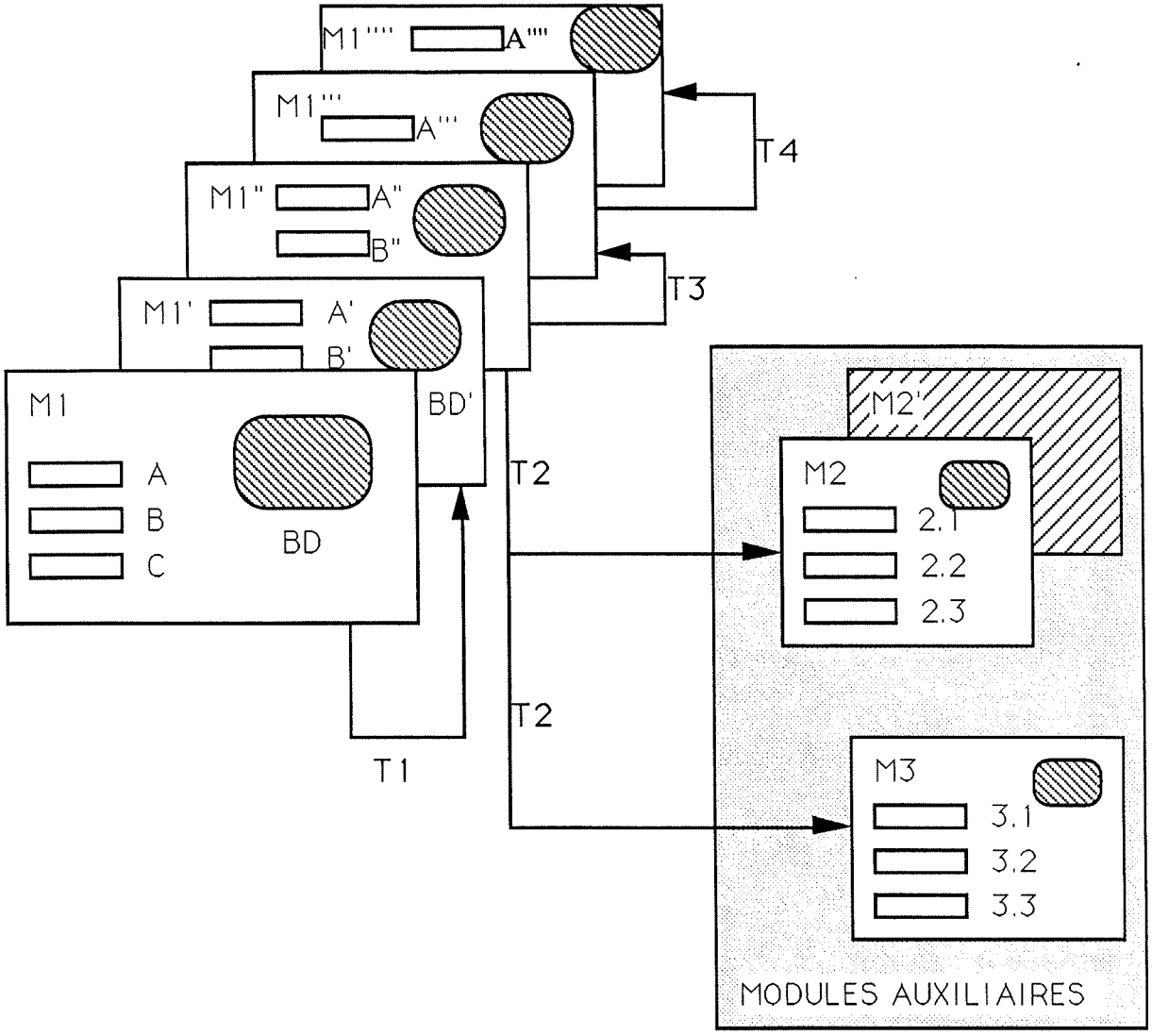
III.2.5 *Schéma récapitulatif*

Illustrons à présent à l'aide d'un schéma l'application de ces différentes techniques de raffinement mises en oeuvre pour le développement d'une machine-phase.

La machine de départ sera constituée par la machine-phase obtenue après la première transformation évoquée au chapitre II, auquelle on aura appliqué ou non les techniques de transformation de signature.

Les techniques de raffinement de machine-phase se combineront sans ordre strict d'application afin d'obtenir le résultat désiré, chaque application d'une technique représentant une nouvelle étape (une nouvelle version des spécifications techniques) dans l'implantation et le développement d'une machine-phase. Ces étapes peuvent être liées au sein d'un arbre de dérivation où le principe du retour en arrière d'une étape à une autre est d'application.

Examinons cela à l'aide de la Figure III.1.



Première technique (T1) : Changement de représentation de la mémoire
Deuxième technique (T2) : Utilisation de modules auxiliaires
Troisième technique (T3) : Mise sous forme algorithmique
Quatrième technique (T4) : Traduction en code exécutable

Figure III.1 :
Illustration des différentes techniques mises en oeuvre dans la conception et l'implantation d'une machine-phase:

La machine-phase M1 constitue le "moteur" du processus d'application des techniques et elle représente en quelque sorte la spécification technique de premier niveau de la machine-phase.

Chaque version de cette machine (M1, M1',...) est décrite par sa mémoire (BD, BD') et ses procédures (A,B,C,A',B',...), les paramètres de celles-ci et les actions dynamiques étant comprises dans cette description.

Un point est à noter : quand on affirme qu'une machine-phase possède de nouvelles procédures (exemple : A",B", C"' au lieu de A',B', C'), il faut préciser qu'en fait ce sera la spécification technique des traitements de ces procédures qui sera modifiée (il se peut qu'aucun changement n'ait eu lieu pour certaines procédures bien sûr), l'objectif et les paramètres de chaque procédure restant toujours identiques pour chaque machine raffinée.

La figure III.1 montre de façon simpliste l'application de chacune des techniques et cela dans un ordre quelconque:

La technique de changement de représentation (**T1**) est appliquée à la machine-phase de départ M1 qui devient la machine-phase M1' après transformation, la mémoire BD étant transformée en mémoire BD' et les procédures A,B,C étant transformées en procédures A',B',C'.

La technique d'utilisation de modules auxiliaires (**T2**) est appliquée à la machine-phase M1' qui est représentée après transformation par la machine-phase M1" (possédant la même mémoire BD' et de nouvelles procédures A", B") et les modules auxiliaires M2 et M3 (le dessin montre par la présence du module auxiliaire M2' la possibilité d'application des mêmes techniques aux modules auxiliaires nouvellement créés).
Les modules auxiliaires M2 et M3 seront spécifiés de la même façon que les machines-phase (ce sont tous des modules informatiques) puisqu'ils sont constitués chacun par une mémoire propre et respectivement pour M2 et M3 par les procédures (2.1, 2.2, 2.3) et (3.1, 3.2, 3.3).

La technique de construction des algorithmes (**T3**) est appliquée à la machine-phase M1" qui est représentée après transformation par la machine-phase M1"' (avec la même mémoire BD' et les procédures A"',...).

La technique de traduction en code exécutable (**T4**) est appliquée à la machine-phase M1"' qui est représentée après transformation par la machine-phase M1"" (avec la même mémoire BD' et les procédures A"",...).

Une proposition d'ordre d'application de ces différentes techniques de raffinement sera étudiée au chapitre 5 mais l'application de telle ou telle technique est guidée en général par les désirs et les objectifs du programmeur. Selon qu'il veuille une réduction des traitements complexes au sein des procédures, une diminution du nombre de variables locales, un regroupement de traitements ou tout autre souhait, il emploiera une certaine technique. D'après le résultat obtenu, il reviendra en arrière ou il considérera d'autres techniques de transformation.

III.3 Architecture globale d'une application informatique

L'architecture globale d'une application informatique peut maintenant être définie. Précisons que l'on traite du cas des applications informatiques intéractives et que l'on se situe bien sûr au niveau des spécifications techniques.

Trois objectifs primordiaux sont à respecter dans la conception de cette architecture :

- **Objectif d'autonomie** entre les deux branches principales d'une application informatique : les traitements (appelées globalement le module informatique) et le dialogue (l'interface Homme-Machine) . Cela permettra principalement un meilleur partage du travail de développement et une efficacité optimale lors de la maintenance de l'application. Notons qu'un effort allant également dans ce sens est celui qui vise à assurer une standardisation maximale des échanges entre les différentes parties de l'application informatique.

- **Objectif d'indépendance** de l'application par rapport aux systèmes physiques. On évoque l'idée d'abstraction et d'une architecture découpée en couches partant d'un niveau abstrait, plus conceptuel à un niveau totalement physique, technique. Cet objectif vise évidemment à pouvoir installer facilement l'application informatique dans plusieurs environnements physiques.

- **Objectif de réutilisation des composants.** Il faut pouvoir utiliser au maximum (en introduisant quelques adaptations si nécessaire) toutes les données et traitements existants afin de réduire les efforts de conception et de développement. Cela exige la construction de composants les plus généraux et les plus souples possible.

La figure III.2 présente schématiquement l'architecture de toute application intéractive, telle que proposée dans [BODART,90].

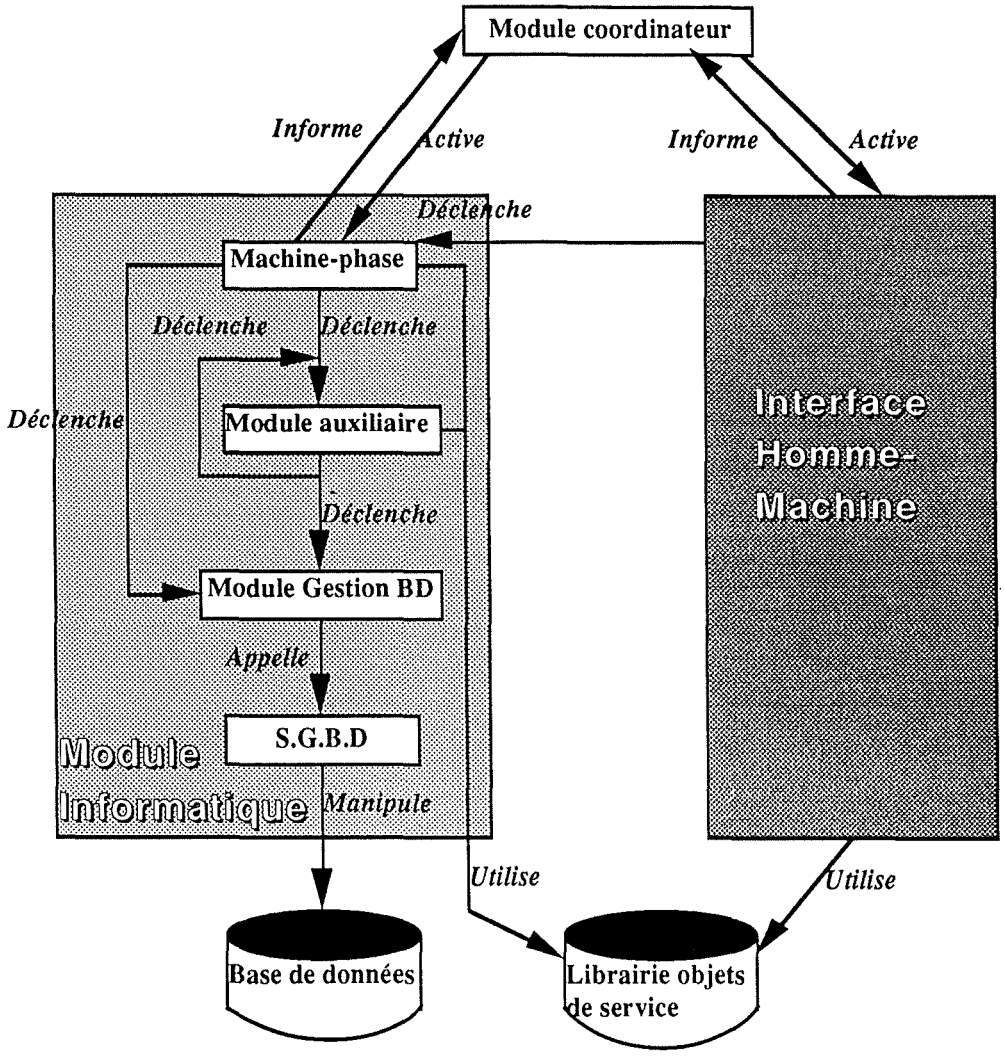


Figure III.2 - Architecture globale d'une application informatique

Le **module coordinateur** s'occupe du contrôle global de l'application et notamment de la coordination des 2 grandes parties: Module informatique (comprenant en fait plusieurs modules informatiques) et Interface Homme-Machine. Le module coordinateur active les services de chacune de ces parties et est informé des résultats obtenus. Généralement, le module coordinateur implémentera les spécifications fonctionnelles du modèle de la dynamique et du diagramme des flux. Son comportement sera d'ailleurs dirigé par ces spécifications. Il s'occupera des échanges de données et il disposera d'une mémoire propre qu'il gérera également.

Ce module coordinateur peut être assimilé également à un module informatique : il dispose d'une mémoire, d'une ou plusieurs procédures et éventuellement de paramètres et actions dynamiques.

La partie **Interface Homme-Machine** n'est pas détaillée ici et ne fera pas l'objet de notre analyse. Signalons simplement que l'interface peut aussi être dérivée systématiquement des spécifications fonctionnelles établies.

La partie **Module informatique** se décompose en plusieurs modules informatiques classés par niveaux.

Au niveau supérieur, on trouvera les **machines-phase** qui déclencheront éventuellement les services de modules auxiliaires (dont les modules de gestion de base de données) pour exécuter leurs fonctionnalités. La relation *déclenche* peut en fait être assimilée aux 2 relations *active* et *informe* car l'objectif commun de ces types de relations est de faire appel à des services en envoyant et recevant des paramètres.

Chaque **module auxiliaire** (pour être vraiment précis on devrait l'appeler module auxiliaire de type fonctionnel par opposition aux modules auxiliaires de type technique tels que les modules de gestion de base de données) peut aussi déclencher les services d'un autre module auxiliaire (fonctionnel ou technique).

Les **modules de gestion de base de données** (niveau logique) , qui se trouvent à un niveau inférieur, appeleront alors le Système de Gestion de Base de Données (niveau physique) pour exécuter leurs différentes fonctions.

Le **S.G.B.D** manipulera évidemment la base de données qui contiendra toutes les informations stockées pour les besoins de l'application informatique entière.

Un dernier point, moins important, est à signaler : une **librairie d'objets de service** se trouvera également au niveau de la base de données et offrira à tous les composants de l'architecture qui le demande (principalement les machines-phase et les modules auxiliaires) des services techniques de base (tri d'ensemble de données, mise à blanc d'éléments, fonction d'ajout d'éléments dans un liste, fonction de passage à l'élément suivant d'un tableau,...). Cette librairie sera en fait la boîte à outils de l'application et constituera un complément des modules auxiliaires en offrant d'autres types de services. La relation *utilise* signifie simplement que le module appelant utilisera tel ou tel service pour remplir son objectif.

Signalons encore que l'interface Homme-Machine, bien que complètement indépendante des modules informatiques définis, pourra déclencher de temps à autre un service d'un de ces modules lors de la validation d'un champ saisi par exemple. Ces services seront très simples et ne feront en général qu'accéder à un élément de la base de données et retourner une valeur (booléenne ou autre). C'est ce que l'on nomme un feedback sémantique.

Cette découpe en niveaux des composants du Module informatique peut être assimilée dans une certaine mesure à une structuration hiérarchique telle que définie au point II.4. Le seul problème qui minimise cette comparaison réside dans le fait que les modules auxiliaires d'un même niveau sont susceptibles de s'appeler entre eux.

L'objectif d'autonomie est assuré par cette architecture car la machine coordinateur gère le comportement de l'application et les échanges entre les deux parties de l'application (sans devoir connaître beaucoup d'informations à propos de celles-ci) et chacune des parties ne doit connaître pratiquement rien des autres pour être développée.

L'objectif d'indépendance par rapport à l'environnement physique est aussi assuré car on a vu que c'est un des buts recherchés par les spécifications des modules informatiques, et les composants physiques de l'application intéractive se trouvent tous au niveau le plus bas de l'architecture.

L'objectif de réutilisation est assuré notamment grâce aux possibilités offertes par les librairies d'objets de service et les modules auxiliaires qui proposent des services généraux et réutilisables selon le contexte.

CHAPITRE IV - APPLICATION DES CONCEPTS ET TECHNIQUES

Ce chapitre a pour but de décrire et d'analyser l'application, dans 2 environnements physiques donnés, des concepts évoqués dans les chapitres précédents.

La machine-phase GestionClients qui a servi d'illustration jusqu'à présent sera implantée dans un premier temps dans un environnement traditionnel comprenant le langage COBOL, le système de gestion de bases de données relationnelles SQL-DS et le gestionnaire d'écrans ISPF, et dans un deuxième temps dans l'environnement constitué par le générateur de code et de transactions DELTA.

Avant toute chose, il faut signaler que les applications développées ne constituent évidemment pas **la** solution unique et optimale (dans le sens de tirer profit de toutes les astuces du langage utilisé afin d'obtenir les plus hautes performances à l'exécution). Les choix qui furent faits sont forcément arbitraires et ont favorisé avant tout l'adéquation d'une solution concrète aux éléments sémantiques développés dans les 3 premiers chapitres.

Signalons que les annexes A et B contiennent respectivement l'application développée dans le premier environnement et l'application développée dans le deuxième environnement.

IV.1 Environnement COBOL / SQL-DS / ISPF

IV.1.1 *Présentation*

Le premier environnement se présente comme suit (les composants de celui-ci étant assez connus, nous ne les détaillerons pas outre mesure) :

- le langage **COBOL** fut choisi pour la conception des modules informatiques et des traitements.

COBOL est un langage complet offrant entre autres un ensemble de structures de contrôle, de possibilités de description et de gestion de fichiers, de fonctions d'édition d'écrans et de rapports, ainsi que des structures de programmes destinés à des fonctions particulières telles que le tri ou la fusion de fichiers.

Un programme COBOL est divisé en parties ordonnées possédant une fonction bien déterminée. Ces parties sont appelées divisions et elles sont composées elles-mêmes de sections comprenant des paragraphes où l'on attribuera la définition des données ou des traitements selon les cas.
Nous verrons cependant qu'une autre structure de programme pourra être aussi utilisée si cela s'avère nécessaire. Le manuel du langage COBOL qui a été utilisé est [CLARINVAL,81].

- la base de données est sous forme relationnelle et le langage **SQL-DS** fut choisi comme langage d'accès et de mise à jour de celle-ci.

Ce langage offre un ensemble de requêtes simples permettant de créer la base de données ainsi que de modifier ou consulter (avec la possibilité de créer des vues de données) ses données.

SQL-DS est un langage quasi-naturel qui utilise un formalisme proche de celui de la base de données pour désigner les objets manipulés.

- l'interface Homme-Machine fut développée à l'aide du gestionnaire d'écrans **ISPF**.

ISPF est un gestionnaire de dialogue fournissant des services à des dialogues durant leur exécution (selon ISPF, une application comprend un ou plusieurs dialogues) .

Les types de services fournis par ISPF sont :

- les services d'affichage
- les services de production/formattage de fichiers
- les services de gestion de variables
- les services de gestion de tables
- autres services

Selon ISPF, un dialogue est décrit comme suit :
un dialogue est une application intéractive fonctionnant sous le contrôle d'ISPF et le développeur de dialogue (c'est à dire le programmeur) devra créer les parties ou éléments de données de ce dialogue.

Chaque dialogue est composé d'au moins une procédure de commande (écrite en REXX ou EXEC2) ou d'un programme (écrit en COBOL), ce que l'on appelle une fonction de dialogue, utilisant des éléments de données qui permettent une intéraction cohérente entre l'ordinateur et l'utilisateur.

Les éléments constitutifs de tout dialogue ISPF sont :

- les fonctions (procédure de commande ou programme appelant les services ISPF)
- les variables (utilisées pour communiquer les informations entre les les différents éléments d'un dialogue et les services ISPF)
- les définitions d'écrans (définissant le contenu et le format de chaque écran)
- les définitions de messages (liés aux écrans)

Tout dialogue n'inclut bien sûr pas nécessairement tous les éléments décrits mais les fonctions et les écrans constituent toujours un minimum obligatoire.

Signalons que l'environnement présenté était implanté sur un mainframe IBM VM/IS 9370.

IV.1.2 *Programmation de la machine-phase GestionClients*

Examinons d'abord d'une façon globale comment la machine-phase GestionClients décrite précédemment et l'architecture de l'application contenant cette machine-phase furent implantées concrètement dans l'environnement choisi (Figure IV.1), toujours en gardant à l'esprit l'objectif de dérivation continue et systématique.

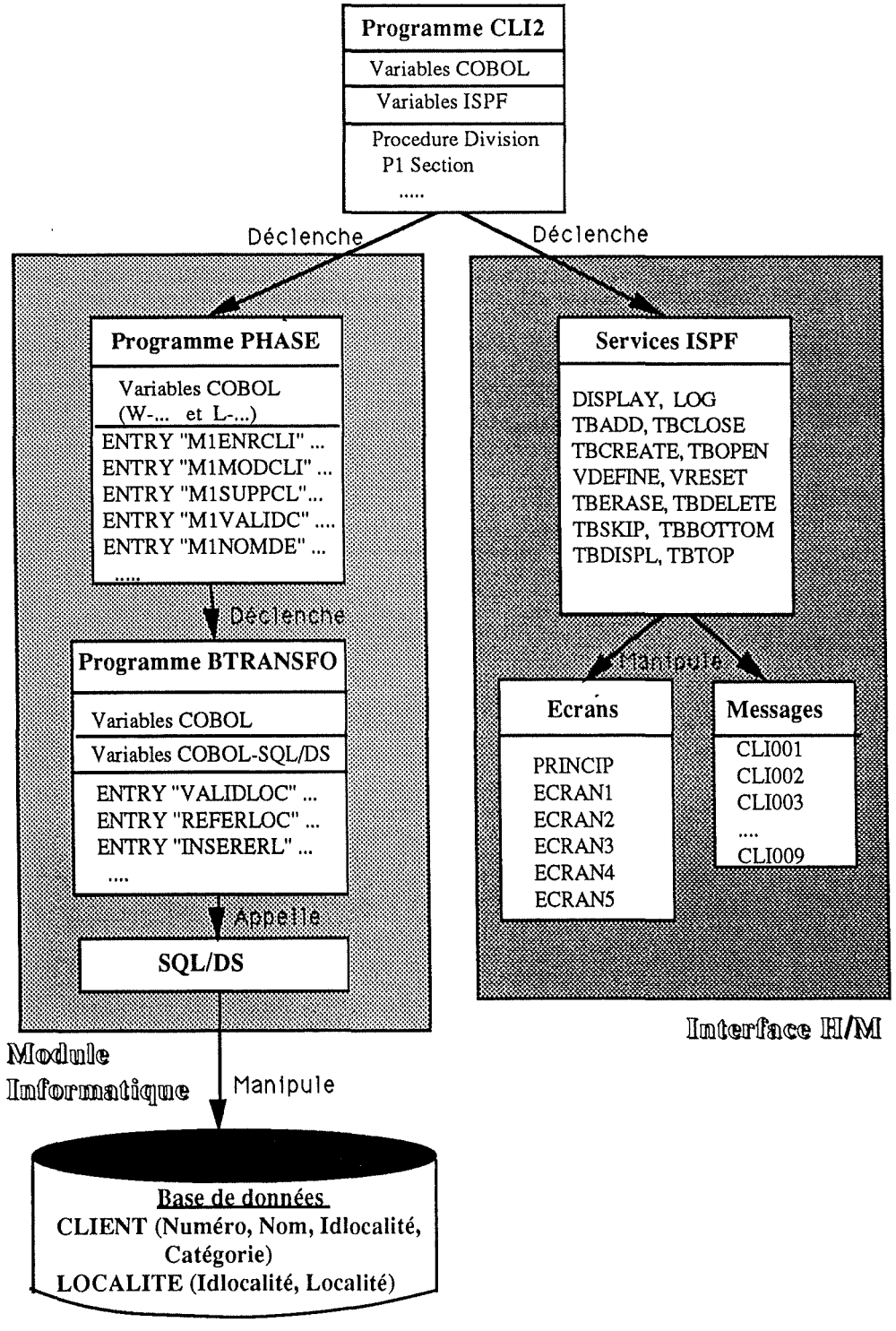


Figure IV.1
Choix de l'architecture d'implantation de la machine-phase en COBOL

Nous avons décidé d'implanter la machine-phase GestionClients ayant subi l'application des techniques de changement de représentation et d'utilisation de module auxiliaire ainsi que les techniques de simplification et réduction de paramètres (techniques de transformation de signature). Nous envisageons donc comme version de la machine-phase GestionClients celle qui utilise les services d'un module auxiliaire technique s'occupant complètement de la gestion de sa mémoire (les 2 relations Clients' et Localites).

Le premier choix effectué fut de représenter chaque module informatique (machine-phase, module auxiliaire,...) par un programme COBOL séparé et non de considérer un seul programme COBOL pour toute l'application (cette dernière alternative va en effet à l'encontre des objectifs assignés à une bonne architecture des traitements).

Examinons cette architecture et la manière dont la démarche de dérivation fut appliquée en COBOL :

Le module coordinateur est représenté par le programme COBOL **CLI2** où l'on définit toutes les variables utilisées lors des échanges de données avec l'interface Homme-Machine et les machines-phase, ces variables constituant en fait sa mémoire.

Les traitements de ce module sont donc destinés à coordonner le comportement global de l'application en appelant au sein des paragraphes définis dans sa structure COBOL les services offerts par l'interface utilisateur (*CALL "ISPLINK" USING ....*) et les services offerts par la machine-phase (*CALL "M1xxx" USING BY REFERENCE* ....). Notons que nous avons omis la spécification précise de ce module et que le comportement de l'application implantée ne s'est inspiré d'aucun schéma précis de la dynamique, cela ayant peu d'importance ici. Le programme CLI2 doit également assurer les transferts de données entre ces différents appels et gérer sa mémoire centrale. On s'aperçoit que le programme CLI2 représente en fait aussi le programme de contrôle du dialogue tel que requis par ISPF. Une autre alternative aurait été de créer un programme COBOL destiné uniquement à l'appel des services ISPF, celui-ci étant alors appelé par le programme coordinateur CLI2.

De toute façon, quelle que soit l'alternative choisie, l'interface Homme-Machine développée à l'aide d'ISPF est construite indépendamment des machines-phase et le module coordinateur ne doit connaître que l'objectif et les paramètres des services et écrans offerts pour fonctionner correctement.

La machine-phase GestionClients est représentée par le programme COBOL **PHASE**.

Chaque procédure et action dynamique spécifiée se retrouve transformée en un point d'entrée bien défini de ce programme (signature de forme : *ENTRY "M1xxx" USING* ...) avec leurs paramètres correspondants (signature: *L-...* , définis en LINKAGE SECTION). Par exemple, le point d'entrée M1ENRCLI représente la procédure MémorisationClient, M1MODLI la procédure ModificationLocalitéClient...

Les traitements COBOL définis dans chacun de ces points d'entrée se dérivent parfaitement des traitements décrits en DESPATH+ et comprennent donc des appels aux services du module de gestion de la base de données et le passage des paramètres nécessaires (*CALL "...." USING BY REFERENCE*).

Nous remarquons cependant que dû à l'utilisation de SQL-DS et à l'existence du module auxiliaire la mémoire de la machine-phase n'a pas dû être explicitement définie dans ce programme (à part les définitions des variables locales de nom W-...) .

Signalons aussi que les contraintes strictes du langage COBOL ont exigé des dénominations différentes des paramètres (la notation des paramètres employée dans la spécification de la machine-phase est réalisable mais a posé quelques problèmes dans l'environnement implanté) et des transferts de données supplémentaires entre ceux-ci.

Le module de gestion de la base de données est représenté par le programme COBOL **BTRANSFO**. Tous les problèmes de gestion de la base de données sont donc rejetés dans cet unique programme. En tant que module auxiliaire, la description des éléments est la même que pour la machine-phase. Les procédures et actions dynamiques sont en effet décrites en COBOL également par des points d'entrée bien délimités dans le programme (avec toujours la définition des paramètres nécessaires en LINKAGE SECTION). Nous avons par exemple le point d'entrée VALIDLOC qui implémente la procédure de validation du nom de localité, REFERLOC qui implémente la procédure de recherche de l'identifiant d'une localité donnée...

Les traitements figurant dans ces points d'entrée, comprendront des instructions COBOL et des requêtes SQL/DS (définies entre les mots EXEC SQL et END-EXEC). Tous ces traitements se dérivent aussi facilement des spécifications techniques.

La mémoire de ce module se limitera à la définition de ses variables locales puisque la description des relations CLIENT et LOCALITE formant la base de données se trouve séparée de tout programme COBOL.

L'utilisation du langage SQL-DS a provoqué cependant dans ce programme quelques remaniements dûs à l'intégration avec le langage COBOL. Il a fallu en effet définir les variables assurant l'interface entre COBOL et SQL-DS.

Au niveau de l'implantation de la mémoire de la machine-phase, dans le cas où l'on utilise aucun module de gestion de la base de données, SQL-DS offre un avantage important au niveau du codage. En effet, avec SQL-DS la mémoire de l'application (et donc de la machine-phase) est définie séparément de tout programme COBOL une fois pour toutes. Cela évite de devoir définir les fichiers utilisés par un programme chaque fois au début de celui-ci comme ce serait le cas si l'on avait choisi de définir la base de données de l'application à l'aide de fichiers COBOL. Dans ce cas, chaque programme représentant une machine-phase ou un module auxiliaire verrait la mémoire de celle-ci décrite explicitement à chaque fois ce qui n'est pas vraiment pratique.

Dans le cas de SQL-DS, il suffit de définir une fois pour toutes la mémoire de l'application (ce qui est très avantageux pour la maintenance) et chaque programme effectuera les opérations sur la partie de mémoire qui l'intéresse.

IV.1.3 *Evaluation de l'environnement du point de vue de la démarche proposée*

Les traitements

Le langage COBOL permet d'assurer une dérivation parfaitement continue des spécifications techniques (et donc fonctionnelles) d'une application informatique vers celle-ci.

Tout module informatique se traduit en général par un programme COBOL et ses composants possèdent très clairement un correspondant COBOL:
la mémoire est implémentée par des fichiers et variables COBOL ou par une autre représentation si l'on utilise un gestionnaire de base de données (comme dans notre cas avec SQL/DS);
les paramètres sont implémentés par des variables COBOL dont la structure reflète parfaitement celle des paramètres;
les procédures et actions dynamiques sont implémentées par des points d'entrée ou des programmes COBOL;
les traitements exprimés en DESPATH+ sont facilement transformables en instructions COBOL (disponibilité de structures de contrôle, requêtes SQL/DS pour les opérations de gestion de la base de données,...).

Le problème principal rencontré réside dans le fait que la structure ordonnée classique des programmes COBOL (*Division, Section, Paragraphe*) ne correspond pas à la structure fonctionnelle des modules informatiques.

L'interface Homme-Machine

ISPF assure une indépendance maximale entre les éléments de données constitutifs du dialogue qu'il permet de créer et les traitements spécifiés en COBOL ou en SQL/DS. Ces éléments et les services qu'offrent ISPF ne dépendent en effet nullement de l'architecture et de la dynamique des traitements qui ont été établies et le module coordinateur se contente de faire appel, au sein de ses traitements, aux services dont il a besoin sans devoir connaître leur fonctionnement interne.

Techniques de transformation de signature

Nous avons pu constater à travers les illustrations du chapitre III que le langage COBOL permettait d'appliquer très facilement et quasi-automatiquement les différentes techniques de transformation de signature.

La correspondance COBOL des paramètres des modules informatiques est évidente et elle respecte pleinement l'objectif de continuité de la démarche.

Les conséquences impliquées par l'application de ces techniques sont aussi facilement dérivées en COBOL (construction de procédures de consultation, création d'une mémoire temporaire,...)

Techniques de raffinement de module

La technique de changement de représentation est facilement applicable dans l'environnement formé par COBOL et SQL/DS. Il suffira de modifier la représentation de la base de données décrite (sous forme de relations) et de mettre à jour les instructions COBOL et SQL/DS des procédures concernées, conformément aux nouvelles spécifications techniques.

La technique d'utilisation de modules auxiliaires ne pose pas plus de problèmes. La transformation en COBOL des modules auxiliaires ne présente aucune autre difficulté que celles rencontrées pour les machines-phase et il suffira d'inclure des appels supplémentaires à ces nouveaux services (points d'entrée) dans les traitements COBOL des procédures concernées.

La technique de mise sous forme algorithmique devra évidemment être appliquée avant tout codage en COBOL étant donné que ce langage est uniquement algorithmique.

La technique de codage est, comme nous l'avons vu, facilement applicable étant donné la simplicité des transformations à effectuer pour obtenir l'application COBOL à partir des spécifications techniques.

Conclusion

Les spécificités des différents composants de ce premier environnement permettent de garantir la continuité de la démarche de dérivation en autorisant des transformations simples et qui respectent parfaitement la sémantique des concepts et techniques proposées.

Cet environnement assure une correspondance parfaite avec les spécifications techniques et la réalisation d'une application se résume pratiquement à une traduction "littérale" de celles-ci.

IV.2 Environnement DELTA

IV.2.1 *Présentation*

Selon la société conceptrice de DELTA (DELTA Software Technologie A.G) , " le générateur de transactions DELTA constitue un ensemble intégré d'outils logiciels destinés à améliorer le développement des applications de gestion " [-,86a] .

L'environnement DELTA est composé de plusieurs générateurs (appelés aussi processeurs) permettant, à partir d'un langage source DELTA, de construire le programme adéquat soit en langage COBOL, soit en langage PL/1. Cela sera effectué en insérant éventuellement à ce processus de génération un macroset (ensemble de macro-instructions) système & utilisateur qui permettra d'adapter parfaitement le programme à la configuration physique choisie et aux désirs de l'utilisateur.

Notons tout de suite que DELTA est indépendant de toute configuration physique, celle-ci devant uniquement être spécifiée et décrite en temps voulu.

" Les améliorations principales attendues par les utilisateurs de DELTA portent en général sur les domaines suivants : maintenance , portabilité , adaptabilité , qualité et stabilité.

Pour obtenir cela, DELTA se fonde sur des méthodes de développement éprouvées :

* structuration du code
* structuration de la logique générale des programmes
* construction d'applications indépendamment des accès physiques
* analyse et documentation structurées
* réutilisation des standards par l'utilisation de librairies de modules " [-,86a]

DELTA est un générateur de COBOL, modulaire, dont les instructions découlent directement des résultats de l'analyse. Il permet de généraliser la standardisation de la programmation et comporte des outils de gestion de la documentation.

Le langage source de DELTA est constitué de commandes, de descriptions de format d'entrée / sortie, de code structuré et de commentaires. Il peut également contenir des instructions en COBOL (très souvent d'ailleurs).

IV.2.1.1 Les processeurs

DELTA utilise plusieurs processeurs afin de permettre la génération d'un programme COBOL. Etant donné leur importance dans l'environnement, nous allons les expliquer d'une manière suffisante :

- DELTA / PROG	(processeur principal)
- DELTA / MACRO	(macro-processeur)
- DELTA / SPP	(programmation structurée)
- DELTA / DETAB	(tables de décision)
- DELTA / FILE	(description des fichiers)
- DELTA / FGEN	(description des enregistrements)
- DELTA / GRU	(gestion des ruptures)
- DELTA / SCREEN	(gestionnaire de grilles, d'écrans)
- DELTA / PSD	(langage de description de structures)
- DELTA/ DIALOG	(contrôle des programmes de dialogue)
- DELTA/ OSP	(squelettes de programmes transactionnels)
- DELTA/ FDOC, XDOC, CDOC	(gestion de la documentation)
- DELTA/ REPORT	(génération de rapports)

DELTA / PROG :
Le processeur DELTA/PROG permet de définir les principes généraux de programmation avec DELTA.

" DELTA/PROG permet de rassembler la définition de l'organisation des traitements, le code propre à chaque traitement et le squelette standard pour constituer un ensemble cohérent conforme au langage de programmation utilisé. " [-,86a] .

DELTA / MACRO :
" La qualité et la fiabilité d'une application ne peut être obtenue qu'avec l'utilisation de standards.
DELTA/MACRO permettra de définir des modules de traitements standards que l'on pourra utiliser dans tout programme. Une bibliothèque de développement dans laquelle les éléments de code (composants) peuvent être modifiés dynamiquement à l'aide de paramètres au moment de leur appel par chaque programme sera mise à la disposition du programmeur. " [-,86a]

DELTA/MACRO interprétera et exécutera les instructions qui peuvent contrôler la génération du code en fonction de paramètres.

DELTA / SPP :

Ce processeur est un outil permettant de coder les traitements de manière structurée à l'aide d'un pseudo-code.

Ses fonctions sont rassemblées comme suit :

- analyse et vérification de la structure du code source
- conversion en COBOL du code source
- génération d'aides à la mise au point

Suite à son appel, on pourra donc utiliser toutes les instructions classiques de la programmation structurée (DO WHILE, IF THEN ELSE, PERFORM, CASE,...) et la plupart des instructions COBOL.

DELTA / DETAB :

Les tables de décision constituent un outil puissant pour décrire clairement les problèmes complexes de traitements interdépendants.

DELTA prendra en charge ce problème particulier grâce au processeur DETAB qui peut être utilisé à la fois au niveau de l'analyse (mise en évidence de leur structure logique, étude de complétude) et de la programmation (génération du code exécutable correspondant).

DELTA / FILE :

Il est primordial que dans chaque programme les données soient définies de la même manière et que les méthodes d'accès soient standards.

L'uniformité dans l'utilisation des données est ici assurée par une séparation des programmes et des accès aux données. Cette séparation - représentée par le concept de "dissimulation" des données - est supportée par l'outil logiciel DELTA / FILE.

Le processeur FILE est un outil permettant de décrire les données au niveau logique dans un programme source DELTA. Ce programme contiendra seulement la description d'une vue de l'application et donnera les indications nécessaires pour communiquer avec les données externes. DELTA/FILE générera automatiquement tout le code d'accès aux fichiers.

DELTA / FGEN :

" DELTA/FGEN est utilisé pour générer et gérer les descriptions d'enregistrements des différents fichiers. Il peut être utilisé seul ou avec des dictionnaires de données. " [-,86a]

DELTA/FGEN permet la séparation des structures d'enregistrement des programmes qui les manipulent.

DELTA / GRU :

" Les programmes devant traiter des ruptures sont structurés de manière spécifique. La structure des données, et par conséquent celle des traitements, est constituée d'itérations imbriquées.

Les conditions de répétition des différentes itérations se déduisent des zones du fichier, chacune de celles-ci se rapportant à l'un des niveaux de rupture à traiter. Dans ce cadre, le processeur GRU permet de définir une structure de programme, automatise la gestion des ruptures, génère une trame standardisée du traitement, avec une structure optimisée et des noms standardisés de locations et reste compatible avec les autres générateurs du système. " [-,86a]

DELTA / SCREEN :

" Ce processeur est destiné à supporter l'intégration, la définition et le développement de masques de saisie (écrans) dans les programmes intéractifs.

Ces écrans de saisie sont développés étape par étape et suivent le principe de dissimulation des données. " [-,86a]

On y spécifie les formats d'édition, les attributs des zones et les règles de validation et d'affichage de l'écran.

La stricte séparation du niveau fonctionnel de l'implantation permettra au programmeur de travailler sans connaître les détails techniques car la connaissance de ces détails est contenue dans les macros du processeur SCREEN.

Un ensemble de macros de dialogue est disponible et celui-ci fournira une série de primitives de manipulation d'écrans qui assurent une interface portable dans de nombreux environnements.

DELTA / PSD :

" DELTA/PSD se base sur le principe utilisé par de nombreuses méthodes de développement de logiciel, à savoir la déduction de la structure des traitements à partir de la structure des données traitées par le programme.

Le générateur PSD est ainsi basé sur un langage non-procédural permettant de définir la structure d'un programme à partir des descriptions Warnier ou Jackson. Cette syntaxe permet une représentation plus condensée et plus claire que l'utilisation de pseudo-code.

DELTA/PSD peut être utilisé dès le stade de l'analyse pour définir les squelettes de programmes batch ou transactionnels. " [-,86a]

DELTA / DIALOG :

" DELTA/DIALOG fournit les macro-instructions nécessaires pour traiter, envoyer, recevoir, valider et stocker les données d'écrans définies par DELTA/SCREEN. DELTA/DIALOG fournit les instructions et mécanismes destinés au contrôle des programmes de dialogue et à la gestion du transfert de données entre les étapes d'un même programme et d'un programme à un autre. " [-,86b]

DELTA / OSP :

DELTA/OSP est une famille de générateurs qui génèrent automatiquement et gèrent des squelettes de programme pour les programmes transactionnels. Chaque générateur est spécialisé dans un type de programme ou transaction particulier :

- OSP-MENU : s'occupe de la gestion des menus
- OSP-BROWSE : s'occupe de la recherche et du listage des données sans connaître exactement la clé de recherche
- OSP-UPDATE : s'occupe de la mise à jour des ensembles de données
- OSP-REFERENCE : s'occupe de l'affichage des données
- OSP-DATA ENTRY : s'occupe de la saisie de nouvelles données

Tout programme source DELTA se basera donc presqu'exclusivement sur les appels à ces processeurs afin d'engendrer l'application voulue. Ces processeurs pourront et devront bien évidemment s'appeler entre eux.

Schématisons simplement l'utilisation de quelques-uns d'entre eux en présentant les principes transactionnels qui sont de mise avec DELTA (Figure IV.2) en vue de développer des applications intéractives.

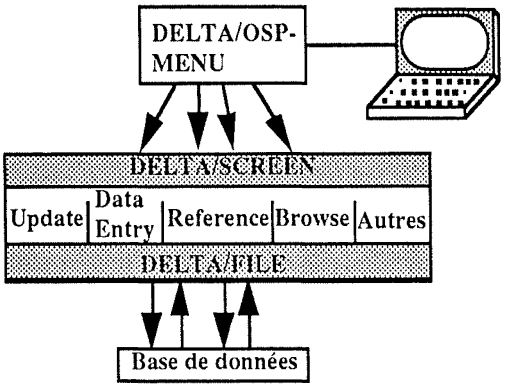


Figure IV.3 - Utilisation des processeurs pour la gestion de transactions

DELTA/FILE assure la gestion de la base de données et l'interfaçage avec les programmes qui l'utilisent. DELTA/SCREEN assure le développement et la gestion des écrans utilisés par les squelettes de programme générés à l'aide des différents processeurs OSP (Update, Data Entry, Reference...). DELTA/OSP-MENU permettra de contrôler l'enchaînement de ces écrans et programmes en construisant un arbre de menus.

IV.2.1.2 La gestion de données

Un autre point important à connaître quant à DELTA est celui concernant les différentes zones de stockage de données qui sont manipulées par les programmes transactionnels dans leurs relations entre eux et avec l'interface utilisateur.
Le flux des données manipulées par tout programme transactionnel DELTA est présenté dans la Figure IV.3 .

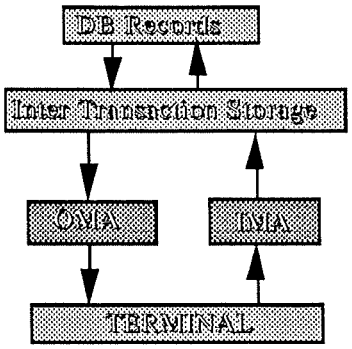


Figure IV.3
Zones de stockage des données en DELTA

L'Inter Transaction Storage est une mémoire d'exécution (physiquement un buffer) qui conserve par défaut son contenu entre chaque transaction, entre chaque étape d'un même programme et entre l'exécution de plusieurs programmes.

Les 2 zones OMA (contenant les données à afficher sur écran) et IMA (contenant les données saisies) jouent en fait le rôle de terminal logique et d'intermédiaire avec le terminal physique, ce dernier n'intervenant jamais dans la construction d'un programme DELTA. Les données de ces 2 zones résident physiquement dans une zone de la mémoire centrale.

Il faut bien sûr aussi mentionner la base de données qui contient la description des fichiers et leur contenu, cette base de données étant le plus souvent remplie par les données de l'ITS.

Le programmeur pourra transférer ces données d'une zone à l'autre en faisant appel à des macros standards ou en codant lui-même les instructions nécessaires.

IV.2.1.3 Quelques définitions

Avant d'aller plus loin dans notre étude de DELTA, il serait bon de fournir quelques explications concernant les concepts principaux qui sont à la base de la programmation en DELTA.

Nous allons définir les notions DELTA suivantes :
- processeur
- programme
- location
- squelette de programme
- macro
- routine

PROCESSEUR : Outil fourni par DELTA afin d'aider l'utilisateur dans sa tâche de programmation. Chaque processeur s'occupe d'un aspect particulier de la programmation et offrira selon les cas, des squelettes standards de programme, des macros ou des routines.

PROGRAMME : Texte en code source constitué de la définition de l'organisation des traitements, de la définition des fichiers à manipuler et du code spécifique aux traitements élémentaires qui ont été définis. Le nom d'un programme DELTA est spécifié lors de l'appel au processeur PROG, première instruction d'un programme.

LOCATION : Point d'entrée défini dans un programme, destiné à accomplir une fonction précise et possédant un nom standard. Les locations peuvent être représentées par des "boîtes" définies dans le texte du programme, dans lesquelles l'utilisateur peut attribuer du code. Le principe des locations permet d'écrire les programmes en regroupant le code par fonction, de structurer ce programme comme étant un ensemble d'entités fonctionnelles, ce qui va faciliter aussi bien le développement que la maintenance des programmes ainsi construits (exemples : location GETRDB destinée à la consultation de la base de données, location CHECK destinée à la validation de données saisies,...)

SQUELETTE DE PROGRAMME : Structure de l'ensemble ou d'une partie du programme définissant la logique générale de celui-ci. Cette structure permet de déterminer l'enchaînement des locations et donc de définir des points d'entrée dans le programme. Les squelettes de programme sont la plupart du temps générés par les processeurs PROG, PSD, GRU, FILE et OSP.

MACRO : Elément de la librairie de code DELTA contenant des instructions du langage de programmation de la machine cible et/ou des instructions du langage macro de DELTA. Ces macros peuvent être standards (c'est à dire fournies par DELTA) ou développées par l'utilisateur pour remplir ses propres besoins.

ROUTINE : Fonction ou procédure, ayant une fonction précise, fournie par les processeurs DELTA. A la différence des macros, ces routines ne peuvent être modifiées dynamiquement et sont en général uniquement fournies par DELTA.

IV.2.1.4 La programmation en DELTA

Examinons à présent les caractéristiques générales de la programmation en DELTA en comparaison avec le langage COBOL.

Peu de points communs sont partagés par COBOL et DELTA. Un programme DELTA comprend le plus souvent tout comme pour COBOL une partie FILE (description des fichiers), une partie WORK (description des variables de travail), une partie PROG (description des traitements) et éventuellement une partie LINK (= Linkage section) mais nous verrons que ces éléments sont bien différents de leurs correspondants COBOL, ne fût-ce déjà que par l'absence d'ordre strict de définition dans le langage DELTA.

Ce qui est par contre véritablement commun, c'est le fait pour DELTA de décrire une grande partie de ses traitements primitifs à l'aide d'instructions purement COBOL telles que MOVE, IF THEN ELSE, COMPUTE, DO WHILE, ...

Les variables locales et les enregistrements sont aussi décrits par niveaux comme en COBOL mais les similitudes se limitent, en toute généralité, à ce genre de choses car la gestion de l'interface utilisateur, la gestion des fichiers et l'organisation des traitements sont complètement différents en DELTA.

La différence majeure entre un programme COBOL et un programme DELTA peut être résumée dans le schéma de la Figure IV.4 .

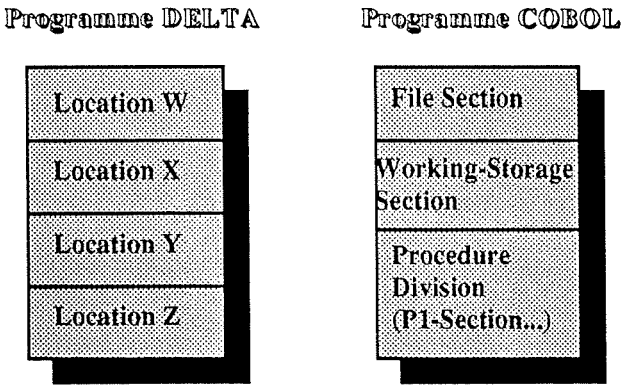


Figure IV.4

Structure des programmes en DELTA et en COBOL

Un programme COBOL possède 3 grandes parties bien fixes et ordonnées qui possédent un objectif précis (nous avons simplifié la schématisation en proposant la Section Fichiers, la Section Variables de travail et la Division comprenant la structure des traitements par sections et paragraphes, omettant volontairement les sections ou divisions moins fréquentes). Un programme DELTA est constitué par contre d'une suite de boîtes fonctionnelles spécifiées sans ordre précis (et complétées selon ses besoins), mais agencées au sein de squelettes standards générés par les différents processeurs.

Analysons à présent brièvement les particularités principales de ce langage.
Trois grandes parties peuvent être dégagées afin d'être clair et complet :

- la description et l'accès aux fichiers et enregistrements
- la description et la gestion des écrans (interface utilisateur)
- le squelette du programme et l'organisation des traitements

Ces 3 points devraient couvrir les particularités principales de DELTA, le but étant ici de garder une certaine généralité.

Description et accès aux fichiers et enregistrements

Comme on l'a dit, l'utilisation du processeur DELTA/FILE offre une séparation des programmes et des accès aux données (concept de "dissimulation" des données).

Les données de la base de données sont donc décrites et implantées séparément de leur utilisation. Le programme connaît seulement l' "enveloppe" des données (le côté application) et peu ou rien au sujet de la façon dont elles sont stockées (renseignement contenu dans la bibliothèque des macros).

Les programmes se contenteront alors de faire appel à des routines d'accès standards telles que PUT (écriture) ou GET (lecture) afin de traiter la base de données comme prévu.

Cette séparation des données du programme permet des accès standards au système de gestion de données, aux bases de données, et même à des "pseudo-fichiers" tables.

Cela constitue un changement de vision considérable par rapport à la programmation "classique" COBOL.

"Classique" car il serait peut-être possible d'arriver à une situation semblable de dissimulation des données en COBOL mais cela exigerait des aménagements et une conception plus complexe.

Dans le cadre de notre démarche de dérivation, cela rencontre le concept de module d'accès aux données auquel ferait appel les modules informatiques car on dispose de services (macros) s'occupant de toute la gestion de cette base de données.

Cette particularité du langage DELTA offre donc de nombreux avantages puisqu'elle garantit une uniformité parfaite du traitement des données et une simplification optimale des accès à la base de données.

Description et gestion des écrans

Le générateur SCREEN permettra de définir des écrans et offrira des routines de contrôle du dialogue aux programmes utilisant ces écrans.
La gestion de l'interface utilisateur est prise en charge automatiquement par les processeurs DIALOG et OSP qui fournissent des squelettes standards de programmes transactionnels et les routines adéquates.

Mais l'utilisateur peut définir lui-même ses propres squelettes de programme pour contrôler le dialogue, en utilisant des macros et routines mises à sa disposition dans ce but.

Squelette de programme et organisation des traitements

Les notions de division, section et paragraphe telles que vues en COBOL n'existent pas en DELTA puisque c'est le concept de locations qui est à la base de la structure des divers traitements.

On remarque en fait pour le COBOL et d'autres langages que la structure ordonnée d'un programme ne correspond généralement pas à sa structure fonctionnelle (dans le cas général, car on a vu que l'on pouvait créer des points d'entrée dans un programme COBOL au lieu d'utiliser sa structure classique, plus lourde).
" Les différentes instructions nécessaires à l'exécution d'une fonction sont dispersées dans le programme, de telle manière que chacune des fonctions ne constitue pas une entité bien visible.
La structure fonctionnelle d'un programme est donc cachée par la structure ordonnée de tels langages, ce qui entraîne un certain nombre d'inconvénients dans le développement et surtout dans la maintenance.

Le principe des locations va permettre de résoudre ces problèmes en structurant le programme comme un ensemble d'entités fonctionnelles reliées entre elles. " [-,86a]
Le découpage d'un programme COBOL pourra dès lors être obtenu à présent en utilisant un ou plusieurs squelettes de programme (fournis en général par les processeurs de DELTA) et le concept des locations.
Chaque location a donc une fonction bien précise (validation de données, écriture dans la base de données, traitement principal, routine,...) et des locations supplémentaires peuvent bien sûr être définies par l'utilisateur.
Dans le cas général, DELTA peut générer, à l'aide des instructions PROG, PHASE et FILE, 3 niveaux de structure dans un programme :
- squelette de programme (squelette général)
- squelette de phase (sous-ensemble du programme)
- squelette de fichier (gestion des fichiers)

Cela permet un niveau élevé de standardisation et le code du programme peut ainsi être bien regroupé par fonction.

Citons simplement comme exemple le squelette général de tout programme DELTA (Figure IV.5), présentant les locations les plus courantes que l'on peut retrouver dans un programme généré à l'aide de DELTA/PROG.

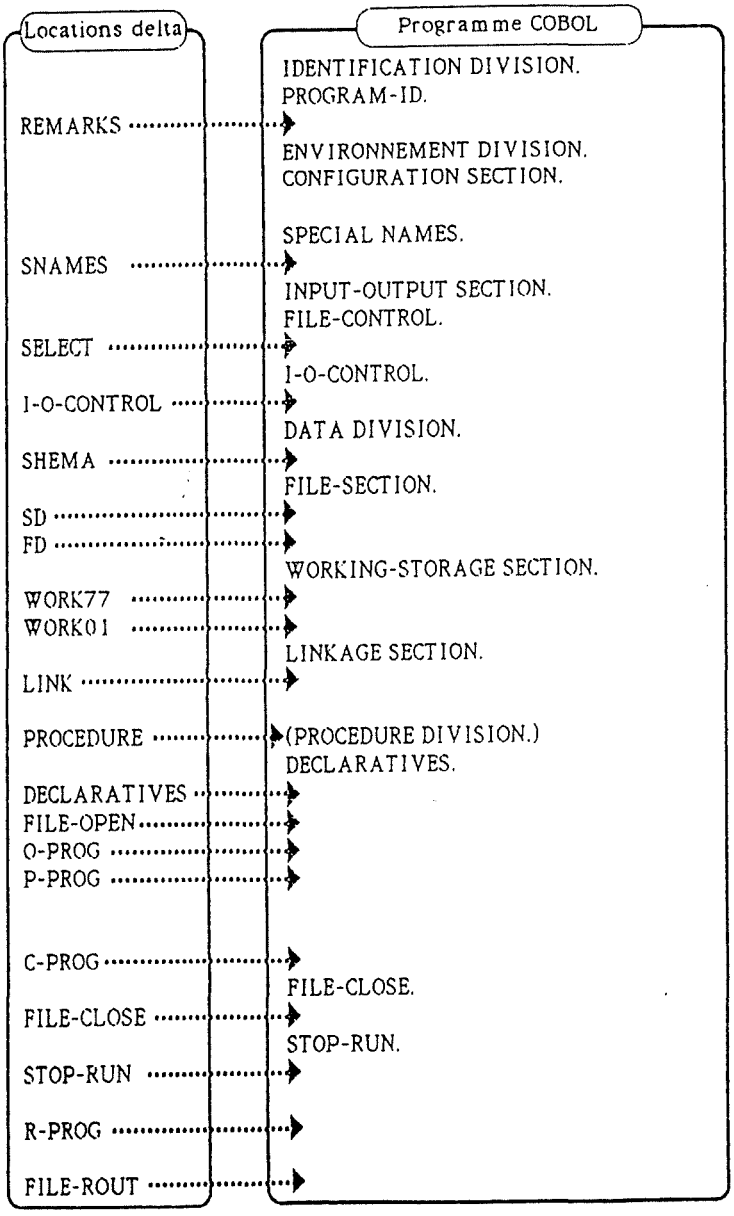


Figure IV.5 - Squelette général d'un programme DELTA

IV.2.2 *Programmation de la machine-phase GestionClients*

Nous avons décidé, pour des raisons de facilité, d'implanter la machine-phase GestionClients dans sa version initiale de spécifications techniques, c'est à dire sans avoir subi l'application de techniques de transformation.

Examinons tout d'abord, à l'aide de la Figure IV.6, l'architecture DELTA de l'application créée. Nous pouvons déjà affirmer que celle-ci semble correpondre plus difficilement (par rapport au premier environnement) à l'architecture type d'une application intéractive telle que dégagée, dans le chapitre III, à partir des concepts utilisés dans les spécifications techniques.

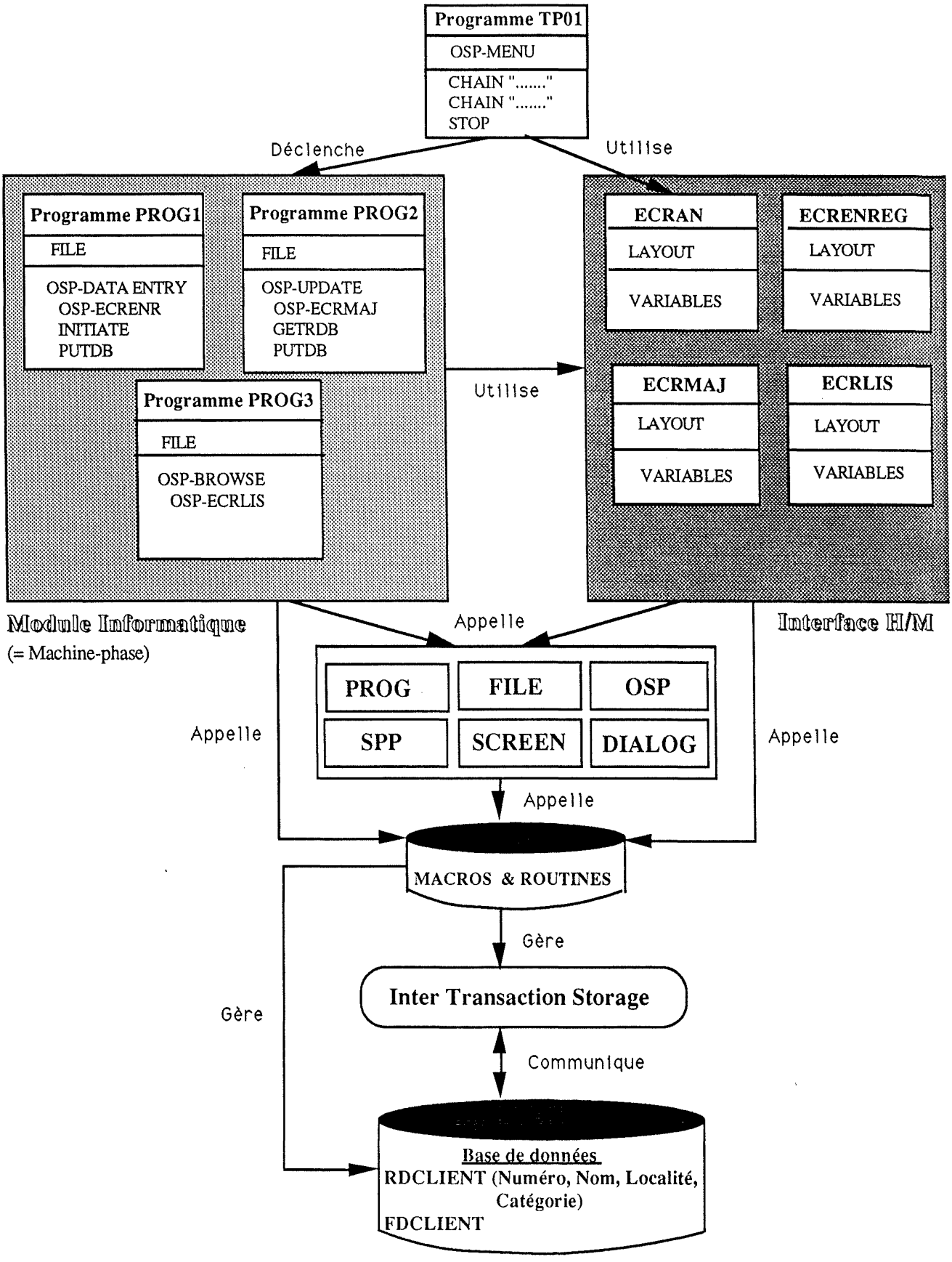


Figure IV.6
Choix de l'architecture d'implantation de la machine-phase en DELTA

La machine-phase GestionClients est implantée comme suit en DELTA :

- sa mémoire est décrite dans les macros FDCLIENT et RDCLIENT et est rendue disponible grâce à l'appel du processeur FILE dans les programmes représentant ses procédures.

- ses paramètres se trouvent définis globalement et une seule fois dans l'ITS et leur valeur est disponible en permanence aussi bien pour l'interface utilisateur que pour tous les traitements de l'application.

- ses procédures sont représentées de la façon suivante :
  * la procédure MémorisationClient est implémentée par le programme PROG1
  * les procédures ValidationClient, SuppressionClient et ModificationLocalitéClient sont implémentées dans le programme PROG2
  * la procédure ListeClientsparCatégorie est implémentée par le programme PROG3

- ses actions dynamiques ne sont pas représentées explicitement car elles sont prises en charge et gérées automatiquement par DELTA

Cette organisation des traitements a été "forcée" par les spécificités de DELTA. En effet, les 3 programmes représentant les procédures de la machine-phase se sont entièrement basés sur les processeurs fournis par DELTA/OSP, processseur spécialisé dans le développement de transactions.
Cet outil correspondait le mieux aux besoins de la machine-phase car il offre des squelettes standards de programme qui traitent quasi-automatiquement les problèmes à gérer par ses procédures.

Détaillons cela : le processeur OSP-ENTRY fournit et gère un squelette standard de programme pour les saisies de données ce qui est l'idéal pour la procédure MémorisationClient; l'appel à OSP-UDATE dans PROG2 fournira un squelette standard pour les mises à jour de données qui représentera les procédures ModificationLocalitéClient, SuppressionClient et ValidationClient; l'appel à OSP-BROWSE dans PROG3 fournira un squelette standard pour la recherche d'éléments sur base d'un (ou plusieurs) critère(s) ce qui correspond idéalement aux fonctionnalités de la procédure ListeClientsparCatégorie. Le programmeur devra alors ajouter des lignes de code dans quelques locations standards de ces squelettes afin d'adapter ceux-ci à ses besoins précis.

Le problème est qu'il a fallu regrouper certaines procédures de la machine-phase au sein d'un même squelette (pour le programme PROG2) et que l'exécution des traitements prescrite par ces squelettes de programme ne correspond pas forcément à la dynamique des fonctions schématisée dans les spécifications fonctionnelles. Néanmoins, ce fut le choix qui a été décidé car cette alternative permettait de profiter au maximum de la puissance de développement offerte par DELTA.

Il eut été toutefois parfaitement possible de représenter chaque procédure de la machine-phase par un programme DELTA séparé (en ne faisant éventuellement pas appel aux processeurs OSP) effectuant uniquement les fonctionnalités prévues de la procédure et se dérivant dès lors parfaitement des spécifications techniques. Mais cela exigerait de nombreuses adaptations et créations de la part de l'utilisateur et du processus de dérivation afin de construire de nouveaux programmes et nous ne profiterions pas des nombreuses possibilités mises à notre disposition par DELTA, ce qui serait stupide.

Remarquons aussi que les écrans de l'interface utilisateur doivent être appelés à l'intérieur de ces squelettes ce qui va quelque peu à l'encontre des principes de l'architecture globale d'une application informatique telle qu'elle a été définie. L'indépendance entre l'interface utilisateur et les traitements de l'application s'en trouve dès lors réduite car le dialogue dépendra des programmes organisant les traitements.

L'interface H/M contiendra la définition des 4 écrans utilisés par les différents programmes créés (ECRAN est utilisé par TP01, ECRENREG par PROG1, ECRMAJ par PROG2, ECRLIS par PROG3).

L'interface échangera les données avec les programmes DELTA le plus souvent par l'intermédiaire de l'Inter Transaction Storage où pourront se trouver toutes les informations utilisées par tous les programmes lors des transactions (ce qui assure une connaissance et une unicité parfaites des données de l'application pour toutes les personnes impliquées dans le projet). Ces écrans seront appelés d'une façon standard à partir des programmes et peuvent bien sûr être créés indépendamment du reste de l'application.

Les programmes utilisant les écrans pourront, à l'intérieur de leur squelette, garnir certaines locations relatives aux écrans appelés (pour la validation de données saisies,...).

Ce qui sert de module coordinateur (ce n'est pas non plus le véritable module coordinateur répondant parfaitement aux spécifications fonctionnelles) dans notre cas est représenté par le programme TP01 contenant en fait la description de l'écran principal c'est à dire le menu général de l'application. Les instructions de ce programme se contenteront d'appeler selon le choix de l'utilisateur un des 3 programmes implémentant les procédures de la machine-phase (à l'aide de l'instruction CHAIN).

Mais le module coordinateur n'est pas une notion fixe en DELTA, selon les spécificités de l'application il pourra être représenté de diverses manières et utiliser plusieurs processeurs de DELTA. Il faut juger cela au cas par cas. En règle générale, un programme principal DELTA construit par le programmeur guidera l'enchaînement des programmes et autres macros en les appelant par diverses instructions et les outils de DELTA

complèteront celui-ci pour assurer le contrôle de l'application. Mais on ne peut vraiment parler de module coordinateur dans cet environnement, l'ensemble des outils proposés par DELTA jouant en fait ce rôle de contrôle des applications.

Cette architecture est également composée de l'ensemble des processeurs utilisés à la fois par les programmes représentant la machine-phase (OSP, PROG, SPP,..) et par l'interface utilisateur (SCREEN, DIALOG,...) pour assurer le bon fonctionnement de l'application.

De ce fait, ces processeurs ne peuvent être classés exclusivement dans l'une de ces 2 grandes parties et ils forment une entité spécifique.

Un ensemble de macros et routines sera également utilisé par l'interface H/M et les modules informatiques afin d'implémenter leurs spécifications. Ces macros et routines peuvent soit être fournies automatiquement par les processeurs appelés, soit être créées par l'utilisateur. Elles géreront (c'est à dire consulteront et mettront à jour) notamment l'ITS et la base de données .

IV.2.3 *Evaluation de l'environnement du point de vue de la démarche proposée*

Les traitements

DELTA permet assez difficilement de dériver d'une façon continue l'application codée à partir des spécifications techniques des modules informatiques.

Nous observons en effet que le fait d'utiliser la plupart du temps des squelettes standards de programme (fournis par divers processeurs) pour exprimer les traitements exige une organisation des traitements et l'introduction d'instructions spécifiques à DELTA et difficilement dérivables des règles de traitement des procédures (par exemple : dans PROG2 : SELECT STATUS-DB ...CASE = DC-STORE ....).

Il semble que ce soit le prix à payer si l'on désire obtenir l'économie de code et la puissance de développement offertes par DELTA.

Il est très difficile d'établir des règles de transformation et de traduction précises et qui soient valables dans tous les cas de figure. Par exemple, une procédure de machine-phase peut être traduite en DELTA, selon le cas à traiter, par un squelette de programme, par une location, par une simple macro, ou même une seule instruction (très rarement). Tout dépend de l'objectif de chaque procédure et de la façon dont on peut l'agencer avec les autres composants déjà définis dans DELTA.

Nous pourrions cependant établir un ensemble de règles conseillant, selon la nature de la procédure à implémenter, l'utilisation de tel ou tel processeur ou macro afin d'aider l'utilisateur dans sa tâche de dérivation (par exemple : utilisation d'OSP pour les procédures constituant une transaction, utilisation de la routine GETR pour une validation de donnée,...).

Un certain nombre de points réglant la démarche de dérivation dans le cadre de DELTA peuvent être établis :

- un module informatique sera traduit en DELTA par un ensemble de programmes (chacun d'entre eux implémentant une ou plusieurs procédures de celui-ci) qui seront découpés en une suite de locations structurées par un squelette de programme créé par l'utilisateur ou généré par un des processeurs appelés dans le programme . Un module informatique sera également traduit par un ensemble de macros standards ou créées par l'utilisateur (une macro pouvant dans certains cas représenter une procédure de machine-phase). Là plupart du temps, la structure de contrôle et la dynamique de ces programmes seront générées automatiquement et l'utilisateur devra seulement ajouter le code d'application.

- les échanges de paramètres entre procédures et modules informatiques seront pris en charge par la mémoire d'exécution de DELTA (l'Inter Transaction Storage). Ces paramètres seront définis une fois pour toute dans cette mémoire et ils seront dès lors disponibles à tous les programmes et écrans désirant les utiliser. Cette mémoire sera gérée par tous les programmes et écrans constituant l'application intéractive ainsi que par les outils DELTA.

- la mémoire des modules informatiques sera décrite dans une ou plusieurs macros et elle sera rendue disponible en spécifiant son utilisation à l'aide du processeur FILE dans chaque programme DELTA l'utilisant, ce qui permet une dérivation parfaite des concepts spécifiés puisque l'on déterminera exactement quelle partie de la base de données (fichier) nous désirons utiliser et selon quelles modalités. FILE offre de plus, grâce à ses routines, une interface standard très efficace entre la base de données et le reste de l'application ce qui assure une uniformisation des accès et mises à jour.

- les règles de traitement seront, selon les cas, une traduction fidèle des spécifications (comme c'est le cas pour PROG1), ou seront difficilement discernables en DELTA (comme pour PROG2). Cela dépendra principalement des squelettes de programme utilisés, et donc des processeurs appelés. Si aucun processeur spécifique n'est appelé (à part PROG bien sûr), la transformation des règles de traitement devrait être quasi similaire à celle vue en COBOL (à l'aide du processeur SPP) c'est à dire continue.

L'interface Homme/Machine

L'indépendance entre la partie dialogue et la partie traitements est réduite en DELTA car :

- les programmes développés, de par l'utilisation des processeurs controlant le dialogue et les traitements, contiennent aussi bien des traitements de gestion de la base de données ou de calcul que des opérations propres à la gestion de l'interface. La séparation entre ces 2 types de traitements est donc annulée.

- le dialogue dépend fortement de la dynamique de fonctionnemment définie dans les programmes. Les écrans, bien que définis séparément des programmes, voient leur utilisation complètement intégrée dans les programmes de traitements et sont dépendants de ceux-ci.

Ici aussi, la continuité de la dérivation de l'interface Homme/Machine à partir des spécifications fonctionnelles semble assez faible puisque le processus de dialogue et la communication entre les écrans et les programmes sont gérés automatiquement par des macros et processeurs, cela afin d'assurer l'uniformité de l'interface.

Techniques de transformation de signature

Etant donné la particularité de DELTA concernant l'implémentation des paramètres des procédures (utilisation d'une mémoire d'exécution), il semble que les techniques de transformation de signature visant à simplifier la structure des paramètres soient moins utiles. L'élimination presque totale des échanges de paramètres entre programmes dans DELTA réduit en effet considérablement leurs avantages .

Par contre, les techniques visant à introduire une mémoire temporaire (élimination de la répétition d'arguments,...) sont tout à fait adéquates et leur application est donc recommandée en vue de faciliter la démarche de dérivation vers l'environnement DELTA.

Techniques de raffinement de module

Technique de changement de représentation :

Cette technique est facilement applicable dans cet environnement. Il suffira de modifier la structure des fichiers et enregistrements concernés (à l'intérieur de leurs macros) et d'adapter dans les programmes utilisant les fichiers modifiés les traitements consultant ou mettant à jour ces fichiers.
Les possibilités offertes par les processeurs FILE et FGEN assurant l'interface avec la base de données permettront de minimiser les changements à effectuer.

Rappelons que la définition des structures d'enregistrement correspond exactement à la définition COBOL de ces données (niveau, nom d'élément et attributs).

Technique d'utilisation de modules auxiliaires :

Nous avons vu que le processeur MACRO mettait des composants modifiables dynamiquement par des paramètres à disposition du programmeur. Ces macros peuvent être définies par le programmeur en y incluant toutes les instructions, paramètres et structures de contrôle nécessaires.

Ces macros sont en fait vues comme des services et la librairie de code qui les contient peut être vue comme un module auxiliaire offrant ces services à tous les composants d'une application intéractive. Ces services sont souvent appelés pour générer du code dépendant de la machine-cible mais ce qui nous intéresse, c'est le fait que ces services peuvent contenir des traitements communs à plusieurs programmes ce qui correspond parfaitement au concept de module auxiliaire fonctionnel. Dans ce cas, remarquons que des passages de paramètres devront être réalisés pour utiliser correctement ces macros.

Le concept de module auxiliaire de gestion de base de données peut être représenté par l'ensemble des macros et routines que le processeur FILE met à disposition des programmes DELTA pour gérer la base de données. Ces macros et routines s'occuperont de tous les détails physiques d'accès et de mise à jour des fichiers décrits.

Signalons que tout comme il existe plusieurs niveaux de modules auxiliaires dans les spécifications techniques , il peut exister également plusieurs niveaux de macros dans l'environnement DELTA (une macro utilise une autre...) .

De plus, nous pouvons aussi construire des programmes assurant les fonctionnalités des modules auxiliaires prescrits dans les spécifications techniques et les faire appeler tout à fait normalement par d'autres programmes. Un programme pourra donc aussi implanter une ou plusieurs procédures de module auxiliaire selon le même mécanisme qu'avec les procédures de la machine-phase.

Cette technique est donc tout à fait applicable dans cet environnement puisque DELTA, de par ses spécificités, se fonde de toute façon sur ce concept de modules et services auxiliaires. De ce point de vue, la continuité de la dérivation est donc parfaitement réalisable (en tenant compte des contraintes de DELTA évidemment).

Technique de mise sous forme algorithmique :

DELTA permet d'exprimer les traitements à effectuer dans un langage proche de l'analyse et donc des spécifications fonctionnelles. C'est le cas avec les processeurs PSD (structure de Jackson), GRU (ruptures) ou SPP (code structuré) mais l'outil le plus utile pour décrire les traitements selon un mode déclaratif est sans aucun doute le processeur DETAB offrant la possibilité d'exprimer les règles de traitement décrites dans les spécifications sous forme de tables de décision.

Cela constitue un avantage considérable pour le processus de dérivation étant donné que la génération des traitements COBOL sera faite automatiquement et l'utilisateur ne devra donc pas se soucier de ce problème de mise sous forme algorithmique.

Cependant, malgré ces avantages, la possibilité de description des traitements sous forme déclarative telle que vue en DESPATH+ ne se retrouve pas en DELTA.

Technique de codage en langage cible :

La génération de code exécutable COBOL ou PL1 est pleinement assurée par les différents outils de DELTA mais comme on l'a dit, la dérivation de l'application en code source DELTA à partir des spécifications techniques n'est pas aussi systématique et continue que nous pourrions l'espérer.

Conclusion

Nous avons pu constater, à travers les différentes parties de cette évaluation, que la facilité d'application de la démarche de dérivation dans l'environnement DELTA était relative, et en tout cas bien moins évidente que pour le premier environnement étudié.

Rappelons brièvement les composants principaux de DELTA qui correspondent le mieux à nos objectifs (et plus précisément aux concepts des spécifications techniques et fonctionnelles) et qui pourront être utilisés au mieux pour assurer la continuité de notre démarche de dérivation (Figure IV.7) :

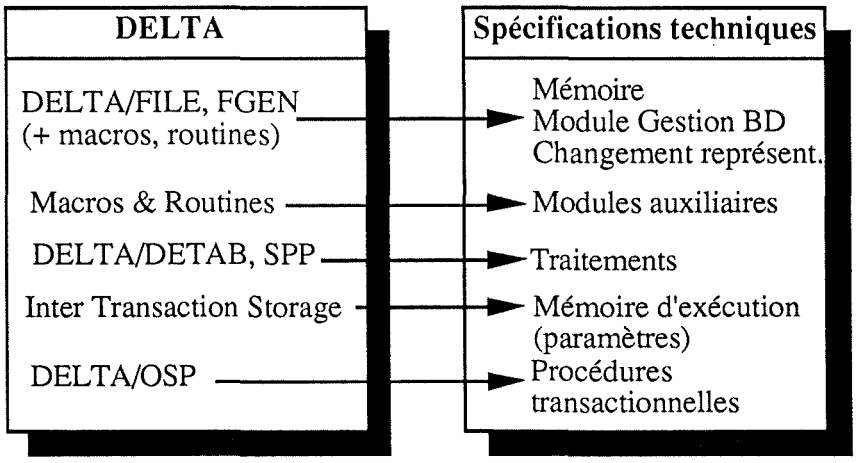


Figure IV.7 - Correspondance DELTA - Spécifications techniques

Les correspondances entre les spécificités de DELTA et les spécifications fonctionnelles IDA sont encore plus difficiles à établir. Il semblerait en fait que pour la démarche de dérivation, seuls les outils DELTA/FILE, DELTA/FGEN, DELTA/DETAB et DELTA/SPP (avec leurs macros et routines associées) puissent être utilisés pour dériver de façon véritablement continue l'application des spécifications fonctionnelles IDA. Les 2 premiers outils seraient utilisés pour implanter physiquement les données du schéma E/A et les 2 derniers seraient employés pour exprimer les traitements dans un formalisme le plus proche possible de l'analyse conceptuelle. Cela n'offre donc rien d'exceptionnel. Tous les autres concepts de DELTA présentent très peu de points communs avec la sémantique d'IDA (spécialement par rapport à ce qui nous intéresse le plus : modèle de structuration des traitements, modèles de statique et de dynamique des traitements) et ne favorisent donc aucune continuité particulière dans la dérivation proposée. Les transformations requises semblent assez complexes étant donné les spécificités offertes par un environnement tel que DELTA.

Une question se pose dès lors : ne vaudrait-il pas mieux, pour des environnements de cette espèce, exprimer les spécifications fonctionnelles selon une autre méthode qu'IDA afin de permettre une dérivation de l'application plus systématique et continue ?

Il semble que cette proposition soit plus que valable. La découpe Application-Phase-Fonction et les autres modèles d'IDA semblent en effet s'éloigner des concepts mis en évidence par DELTA. Ne vaudrait-il pas mieux réfléchir dès le stade de l'analyse conceptuelle en terme de transactions ou de dialogues par exemple ? Dans le cadre de cet environnement (et uniquement celui-ci bien sûr), cela faciliterait assurément la dérivation de l'application informatique et permettrait de profiter au maximum de la puissance de développement et de maintenance offerte par DELTA.

Cependant, on ne peut remettre en cause une méthode de production de spécifications fonctionnelles telle qu'IDA uniquement pour satisfaire les besoins d'un environnement spécifique. IDA présente l'avantage d'être complète, stable, générale et indépendante de toute implantation physique et il faudrait étudier beaucoup d'autres types d'environnements non-traditionnels pour se permettre d'émettre des critiques vraiment fondées à son sujet.

CHAPITRE V - PROPOSITION DE DEMARCHE DE DERIVATION

Ce chapitre a pour but d'exposer une proposition de démarche de réalisation d'une partie de l'application informatique à partir des spécifications fonctionnelles d'une phase.

Cette démarche coordonnera et regroupera en une suite d'étapes précises les différentes techniques exposées au chapitre III.

V.1 Introduction

Cette démarche doit aider le concepteur d'un S.I à élaborer progressivement, dans un environnement de programmation donné, la structure et le comportement du futur système à partir de ses spécifications fonctionnelles validées, en se faisant idéalement aider par un outil logiciel adapté.

La démarche de dérivation devra assurer la continuité dans le développement afin que les applications informatiques soient un prolongement des spécifications fonctionnelles (profitant ainsi de la rigueur et du formalisme qui leur a été apporté) et non une nouvelle construction.

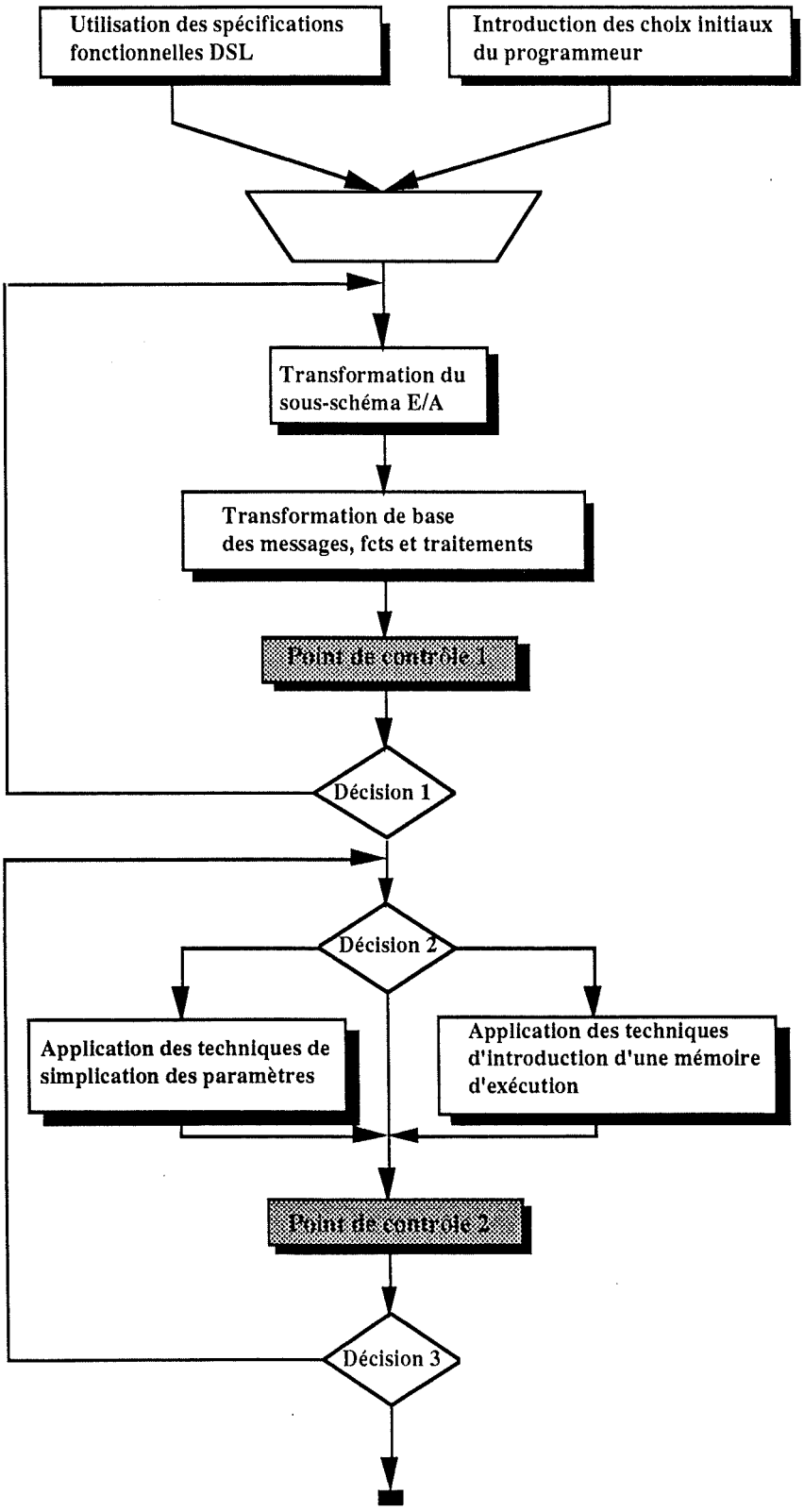
Les caractéristiques principales qu'elle doit posséder sont exprimées comme suit :

- continue
- systématique (règles de construction/transformation précises)
- sans rigidité excessive (il faut conserver une souplesse dans cette réalisation systématique, c'est à dire permettre des adaptations manuelles en plus de l'application des règles en restant toutefois au stade de l'aménagement de la solution standard obtenue)
- correctement documentée (cela étant indispensable à la maintenance des systèmes d'information)

Nous allons donc décrire ces différentes étapes (comprenant elles-mêmes un enchaînement ordonné d'opérations à effectuer) pour lesquelles nous pouvons constater que les adaptations manuelles permises au programmeur sont présentes à différents endroits de la démarche, ce qui rend celle-ci très souple bien qu'elle comprenne le maximum de transformations automatiques.

Cette idée d'approche transformationnelle composée d'une séquence de plus petites étapes est bien plus efficace et plus fiable qu'un type d'approche consistant à appliquer des transformations globales et complexes.

V.2 Démarche de dérivation



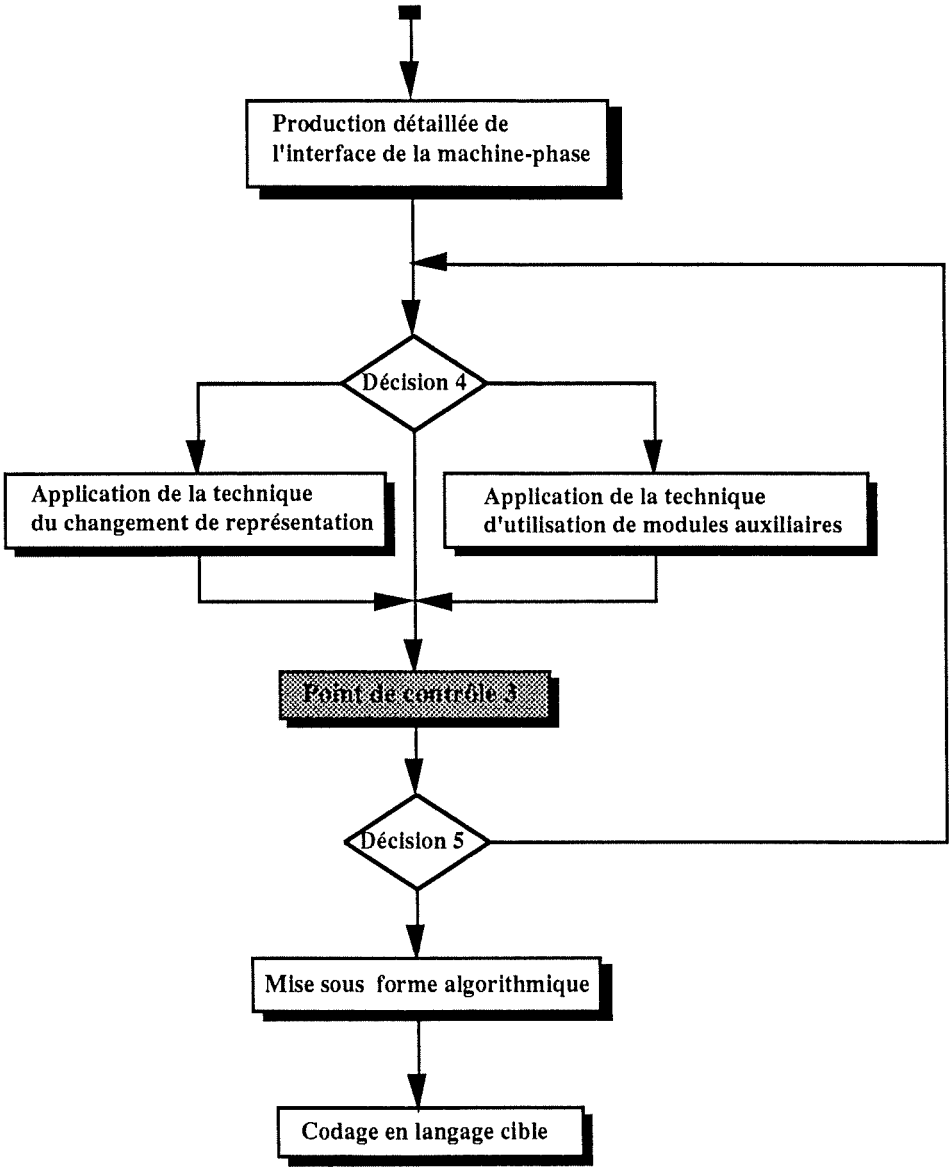


Figure V.1 - Proposition de démarche de dérivation

V.2.1 *Utilisation des spécifications fonctionnelles DSL*

Les spécifications de départ mises à notre disposition seront sous la forme DSL car ce langage permet de représenter le plus fidèlement possible les spécifications fonctionnelles produites selon la philosophie IDA.

Nous supposerons que ces spécifications auront été validées du point de vue de la complétude et de la cohérence, et consolidées.

V.2.2 *Introduction des choix initiaux du programmeur*

Ces choix comprennent :
- des décisions globales valables pour l'ensemble de la démarche
- des choix spécifiques à chaque étape de la démarche.

Pour cette dernière catégorie notons qu'il est possible que des choix soient posés à nouveau au programmeur au cours de la démarche afin d'obtenir plus de précisions dans le contexte de la situation à laquelle on a abouti ou si un problème ou conflit survient (souvent suite à l'exécution des points de contrôle).

Les choix initiaux à poser sont :

- Choix du système cible (configuration physique, langage, SGBD,...). Ce choix est principalement utilisé lors de l'étape de codage mais il sera aussi utilisé auparavant (pour la définition des entête et forme des procédures, forme des paramètres, représentation de la mémoire,...)

- Choix de la technique de représentation des paramètres (application de la simplification, réduction, élimination des paramètres répétitifs, utilisation d'une mémoire d'exécution)

- Premier choix concernant l'application des techniques de représentation de la mémoire et d'utilisation des modules auxiliaires (une plus grande liberté est laissée au départ afin de laisser plutôt le programmeur choisir les techniques de raffinement qu'il veut tester au moment où celles-ci sont effectivement appliquées).

- Autres choix :
  Dans quelle mesure se baser sur les valeurs par défaut ?
  Quel niveau de détail choisir dans la description des traitements ?

V.2.3 *Transformation du sous-schéma E/A*

Cette étape est destinée à obtenir une mémoire de machine-phase exprimée selon un formalisme choisi. On appliquera les méthodes de transformation habituelles et au point de vue de l'outil on pourra faire appel à TRAMIS pour effectuer toutes les transformations voulues .

Le point de départ sera un sous-schéma de données de la phase sous forme E/A classique et le résultat sera la mémoire exprimée souvent selon une représentation E/A qui sera adaptée aux spécificités de l'environnement cible (modèle relationnel,...). Il est évidemment loisible et fréquent de garder la notation E/A originale.

Cette opération de traduction est exécutée à ce stade-ci car c'est une étape particulière ne dépendant d'aucune autre mais qui influencera les étapes ultérieures.

Pour plus d'informations sur les transformations de schémas E/A, il est conseillé de consulter les travaux du professeur J-L Hainaut et notamment [HAINAUT,88].

V.2.4 *Transformation de base des fonctions, messages et traitements*

Les opérations à effectuer seront, dans l'ordre :

- Etablissement des correspondances entre fonctions et procédures (entête de procédure,..)
- Transformation de base des messages de chaque fonction en paramètres (en fonction du langage cible)
- Transformation des contraintes portant sur les messages reçus en préconditions et création des procédures éventuelles de validation de préconditions
- Transformation des conditions portant sur les messages générés en résultats booléens
- Transformation en DESPATH+ des règles de traitement des fonctions
- Création éventuelle des procédures s'occupant des actions dynamiques

Cette étape est bien sûr très importante car il est primordial d'avoir des spécifications techniques de départ correctes sur lesquelles on pourra se baser sans aucun problème. Après l'exécution de cette étape, nous aurons donc une machine-phase sous sa forme initiale brute, c'est à dire une transformation continue d'une phase IDA.

V.2.5 *Application des techniques de simplification des paramètres*

Cette étape permettra au programmeur d'appliquer, au choix, une des techniques de transformation de signature suivantes : réduction de paramètres et simplification de paramètres.

L'étape suivante proposera les autres techniques de transformation de signature à savoir celles qui sont destinées à éliminer les paramètres répétitifs.

Ces 2 techniques sont proposées conjointement car elles possèdent le même objectif et sont très souvent appliquées simultanément. Cela permettra un automatisme plus fort dans la transformation. De toute façon, le programmeur a le choix de n'en appliquer qu'une s'il le désire.

Dans les cas (les plus fréquents) où le programmeur décide d'appliquer ces 2 techniques, les opérations suivantes seront exécutées successivement :

- Réduction puis simplification des paramètres
- Création des procédures de consultation du contenu des paramètres modifiés
  Vérification préalable de leur non-existence
- Reformulation éventuelle des préconditions portant sur les paramètres modifiés
- Modification des traitements des procédures dont les paramètres ont été modifiés:
  + Remplacement des composants des arguments simplifiés par des appels aux procédures de consultation créées
  + Suppression de certains traitements portant sur les composants de paramètres simplifiés ou réduits (selon les règles établies)

La modification qui sera exécutée en premier lieu portera sur les paramètres (puisque c'est la finalité de cette technique) et ce changement produira les autres modifications.

Cette suite d'actions pourra être adaptée facilement si on n'exécute qu'une des 2 techniques.

V.2.6 *Application des techniques d'introduction d'une mémoire d'exécution*

Cette étape permettra au programmeur d'appliquer au choix une ou plusieurs des techniques de transformation de signature suivantes : élimination des arguments répétitifs, élimination des résultats répétitifs et technique générale de la mémoire d'exécution.

Rappelons que cette dernière technique est une généralisation des 2 autres et qu'elle consiste à simplifier au maximum les échanges de paramètres (même ceux non-répétitifs) en gérant pour la machine-phase une mémoire d'exécution complète c'est à dire qui est pratiquement égale à la mémoire de la machine-phase.

Quelle que soit la technique (ou les techniques) choisie(s), les opérations suivantes seront exécutées successivement :

- Construction de la mémoire d'exécution (très souvent sur base de la mémoire existante de la machine-phase) ou transformation des paramètres concernés en mémoire d'exécution

- Mise à jour des paramètres des procédures concernés par la transformation (en général : suppression d'arguments)

- Création des procédures nécessaires de gestion de la mémoire d'exécution : soit les 3 procédures principales du mécanisme itérateur ou accumulateur créé, soit d'autres procédures dans le cas de la technique générale.
  Vérification préalable de leur non-existence

- Modification des traitements des procédures dont les paramètres sont modifiés :
  + Appels aux procédures de gestion de la mémoire d'exécution
  + Suppression de certains traitements portant sur les composants de paramètres qui ont été supprimés

En ce qui concerne la technique générale d'utilisation d'une mémoire d'exécution, on ne devra pas se baser uniquement sur la structure des paramètres pour pouvoir l'appliquer. En effet, il faudra se baser aussi sur les connaissances que l'on a de la dynamique des procédures de la machine-phase et de leurs traitements afin de procéder aux modifications adéquates des traitements et paramètres, et de vérifier leur validité.

V.2.7 *Production de l'interface détaillée de la machine-phase*

Suite à l'exécution de ces premières étapes, on pourra définir précisément l'interface de la machine-phase qui devra être connue par les autres composants de l'architecture globale de l'application afin de pouvoir utiliser correctement les services de la machine-phase.

Cette interface se compose de la déclaration de l'entête de toutes les procédures, de la définition de leur objectif et des paramètres à spécifier et elle comprendra une partie documentaire fournissant les explications nécessaires.

Un document complet sera donc produit et mis à disposition des concepteurs de l'application informatique.

V.2.8 *Application de la technique de changement de représentation*

Les opérations suivantes seront exécutées successivement :

- Modification de la mémoire de la machine-phase :

  + La mémoire sera présentée graphiquement selon le formalisme E/A ou sous une autre forme.
  + L'utilisateur pourra modifier les entités, associations, attributs, rôles,... pourvu que la cohérence et l'intégrité des données soient préservées par rapport au schéma global de l'application. Cela sera vérifié intéractivement par l'outil considéré.

- Modification des traitements des procédures utilisant les objets modifiés du sous-schéma E/A :

  Ces procédures dont les traitements doivent être modifiés peuvent évidemment se trouver dans un module auxiliaire déjà créé.
  Tout traitement modifiant ou consultant un objet du sous-schéma E/A qui a été modifié sera adapté à la nouvelle représentation choisie de la mémoire. La plupart de ces adaptations pourront être générées automatiquement et les valeurs par défaut proposées seront soumises éventuellement à l'approbation du programmeur (exemple: Dans le cas d'un enregistrement des valeurs d'attributs d'une entité, après décomposition de cette entité en 2 entités on enregistrera les mêmes valeurs dans les attributs correspondants de ces 2 entités).

V.2.9 *Application de la technique d'utilisation de modules auxiliaires*

Rappelons que l'on peut distinguer 2 types principaux de modules auxiliaires:

les modules auxiliaires fonctionnels (contenant la plupart du temps des procédures communes à plusieurs machines-phase) et les modules auxiliaires techniques (contenant des services spécifiques à une tâche comme la gestion de la base de données)

Deux possibilités d'utilisation de cette technique peuvent être dégagées :

Cas 1 : Création d'un nouveau module auxiliaire ou d'un nouveau service au sein d'un module auxiliaire déjà existant (l'automatisme de la démarche est évidemment fortement réduit dans ce cas)

Cas 2 : Recherche parmi les modules auxiliaires existants des services pouvant être appelés pour remplir l'objectif fixé

Les opérations suivantes seront exécutées successivement :

- Affichage de la liste des modules auxiliaires existants et de leurs procédures (nom, objectif, modalités d'utilisation, paramètres)

- Choix de la possibilité d'utilisation :

  Dans le cas 1 : Construction du module auxiliaire et des procédures à créer

  Le système effectuera des contrôles de cohérence en permanence, guidera le programmeur en lui proposant des possibilités sur base des machines-phase et modules auxiliaires existants et le préviendra si des similitudes apparaissent avec cet existant.

  Dans le cas 2 : Choix optionnel des services auxquels on fera appel. Si aucun choix n'a été fait, le système appliquant la démarche procédera à une recherche automatique des services potentiellement utilisables, après exécution de l'étape qui suit.

- Indication éventuelle des procédures de la machine-phase que le programmeur considère susceptibles de faire appel aux services de modules auxiliaires (le choix par défaut étant de considérer toutes les procédures de la machine-phase)

- Recherche automatique par le système des similitudes entre les traitements des procédures indiquées et les traitements des services des modules auxiliaires éventuellement choisis

- Remplacement des traitements repérés par des appels aux services de modules auxiliaires adéquats (éventuellement après approbation par le programmeur et adaptation des services)

V.2.10 *Mise sous forme algorithmique*

Il suffira de transformer les groupes de traitements définis de façon déclarative en traitements définis de façon algorithmique. On appliquera les transformations classiques : élimination du parallélisme, utilisation de la notion de boucle, assignation précise de valeur aux éléments contenus dans un POUR .. APPARTENANT A , ...

Cette phase est entièrement automatique sauf en cas de conflit (à résoudre par le programmeur).

Le système passera en revue toutes les procédures pour vérifier cette forme procédurale et appliquera ses règles de transformation.

V.2.11 *Codage en langage cible*

Cette étape sera également complètement automatique et il faudra créer un générateur qui soit valable pour le plus grand nombre possible d'environnements physiques. Pour les environnements cibles complexes (non traditionnels), il semble qu'un générateur spécifique soit nécessaire pour chacun d'entre eux.

V.2.12 *Points de contrôle*

Ces points de contrôle se justifient par le principe de la détection la plus rapide possible d'erreurs dans le processus de dérivation. Ces étapes vérifieront, après l'exécution d'opérations de transformation, les qualités suivantes :

- Non-redondance des éléments de la machine-phase entre eux et avec les autres composants de l'architecture
- Correction syntaxique et sémantique de chaque élément de la machine-phase
- Cohérence globale de tous les éléments de la machine-phase avec l'ensemble des composants de l'architecture

En cas de problèmes, le programmeur sera évidemment prévenu des modifications à effectuer et des retours en arrière aux étapes défectueuses seront possibles.

V.2.13 *Points de décision*

Chaque point de contrôle sera évidemment suivi d'un point de décision offrant trois possibilités :

Soit la continuation de la démarche de dérivation en stipulant éventuellement quelques avis ou avertissements à prendre en compte,
soit le retour à une étape précédente en signalant les problèmes survenus et les solutions proposées (on annule donc tous les effets de la dernière application de technique),
soit une nouvelle itération de l'étape précédente pour pouvoir effectuer une application de technique supplémentaire (en respectant les contraintes imposées).

Le choix effectué lors de ces points de décision est donc déterminé uniquement par les résultats obtenus suite à l'exécution automatique du point de contrôle qui les précèdent. C'est le cas des points de décision 1, 3 et 5.

Le choix du point de décision 2 est déterminé par les choix initiaux introduits par le programmeur et éventuellement par les choix à reposer si un problème se produit. Ce point de décision permettra de sélectionner les techniques de transformation de signature à appliquer afin de pouvoir comparer différentes alternatives.

Le choix du point de décision 4 est également déterminé par les choix introduits par le programmeur . En général, trois scénarios d'application successive des 2 techniques de raffinement proposées sont énoncés :

- Application de la technique de changement de représentation suivie de la technique d'utilisation de modules auxiliaires
- Application de la technique d'utilisation de modules auxiliaires suivie de l'application de la technique de changement de représentation (applicable aussi aux modules auxiliaires créés)
- Application répétitive de la technique d'utilisation de modules auxiliaires (création de plusieurs niveaux de modules auxiliaires)

Mais le cas de base reste quand même celui où l'utilisateur choisit d'appliquer uniquement une de ces deux techniques de raffinement à la machine-phase.

Signalons enfin deux remarques importantes à propos de cette démarche de dérivation :

- Tout au long du processus (entre les étapes et à l'intérieur de celles-ci), des choix pourront être posés à l'utilisateur afin de résoudre des conflits ou introduire les paramètres éventuels et modalités d'application des différentes techniques. Néanmoins, le processus appliquera automatiquement toutes les transformations définies et ne fera intervenir le programmeur que dans les cas vraiment indispensables.

- Chacune de ces étapes produira un rapport de documentation reprenant les choix faits, les transformations effectuées et les problèmes survenus. Cela est indispensable pour la maintenance de l'application informatique produite.

V.3 Conclusion

Cette démarche devra constituer le fondement du processus industriel qui est envisagé et qui permettra d'obtenir systématiquement, en partant des spécifications fonctionnelles d'une application informatique et en désignant un environnement cible ainsi que les techniques de transformation à appliquer, le programme en code exécutable qui implémentera ces spécifications fonctionnelles.

Notons que cette démarche est générale et qu'à ce stade-ci elle possède de nombreuses limites d'application. La limite principale réside dans le fait que son caractère automatique est différemment prononcé selon l'environnement cible que l'on choisit.

Nous avons en effet pu constater que certains environnements non-traditionnels (tels que DELTA par exemple) demandaient beaucoup plus de travail pour obtenir une dérivation continue et automatique des spécifications fonctionnelles (exprimées selon IDA) et que les techniques énoncées ne semblaient pas toutes indispensables étant donné la puissance de programmation et/ou de développement offerte par de tels environnements.

Mais il faudra étudier beaucoup plus d'applications et d'environnements pour avoir une idée certaine à ce sujet et obtenir une démarche affinée et optimale.

Cependant, la démarche proposée qui comprend 4 grands groupes d'étapes à savoir:
- Construction de la base de données
- Transformation de base des spécifications fonctionnelles
- Transformation des paramètres
- Transformation des traitements

peut s'appliquer à la majorité des environnements physiques traditionnels de façon systématique et en continuité avec les spécifications fonctionnelles.

CHAPITRE VI - SUGGESTIONS POUR L'OUTIL LOGICIEL ET SUJETS DE REFLEXION

Ce chapitre final est décomposé en 2 parties. La première partie a pour objectif d'exposer un ensemble de principes généraux qu'il sera bon de prendre en compte lors de la conception future d'un support logiciel au processus de dérivation étudié. La deuxième partie de ce chapitre regroupe quelques questions qui se sont posées dans ce travail et qui mériteront une attention particulière pour la suite des recherches qui seront entreprises.

VI.1 Suggestions pour l'outil logiciel

L'outil à développer devrait répondre aux caractéristiques globales suivantes :

- **ne pas être rigide** en ce qui concerne l'application des transformations c'est à dire proposer plusieurs choix (sans violer les contraintes d'enchaînement logique des étapes) à l'utilisateur après certaines étapes de transformation avec éventuellement la possibilité de retourner en arrière d'une ou plusieurs étapes.
L'outil ne fera en fait que respecter les propositions de la démarche de dérivation qui ont été énoncées dans le chapitre précédent .

- appliquer un maximum de **valeurs et transformations par défaut** (par exemple : dénomination des procédures, type des paramètres, transformation de données,...)

- **être ouvert à différents environnements cibles** (c'est à dire assurer une indépendance maximale par rapport à l'environnement physique)

- s'inscrire au plan ergonomique dans le cadre d'une **approche générale** couvrant aussi bien les transformations de données et de traitements que celles des dialogues c'est à dire intégrer de façon cohérente la transformation des traitements d'une application intéractive avec la construction de la base de données et de l'interface utilisateur.

- l'outil devra toujours garder le **contrôle de la démarche** (connaître la trace d'exécution des opérations en cas de retours en arrière,...) et réagir correctement à toutes les actions de l'utilisateur.

- l'outil devra guider l'utilisateur en lui proposant notamment une **aide permanente** aussi bien active que passive, et en lui signalant et expliquant les conséquences exactes de chacune de ses actions.

- Au niveau de l'interface de l'outil, le multi-fenêtrage et l'interfaçage mixte (parties graphique et textuelle) semblent indispensables pour avoir une vision claire du système et pour évaluer directement les conséquences de ses actions.

VI.2 Sujets de réflexion

VI.2.1 *Mise en oeuvre de la démarche de dérivation*

En plus des diverses propositions évoquées, il faudra prendre en compte les remarques suivantes pour appliquer correctement la démarche de dérivation.

1. Nous avons toujours considéré que nous nous trouvions dans une perspective d'utilisation mono-utilisateur de l'application informatique, tous les problèmes relatifs à une utilisation multi-utilisateur (concurrence pour l'utilisation de procédures, partage, répartition des données, communication,...) ont donc été volontairement ignorés.

2. Un ensemble de règles précises et automatisables gérant l'application des différentes techniques de transformation devra être mis au point et il semble qu'une partie de cet ensemble devra être créé spécifiquement pour chaque langage cible choisi.

3. Nous n'avons pas encore évoqué d'ordre précis de transformation des fonctions figurant dans les spécifications fonctionnelles. Afin de profiter au mieux du travail de transformation réalisé, il serait utile de se donner une ligne de conduite.

Cinq ordres de prise en compte des fonctions peuvent être proposés :

- Ordre par importance sémantique ou conceptuelle des fonctions (par exemple, dans une phase d'enregistrement de commandes, on trouverait : 1° EnregistrementCommande, 2° ValiderCommande, 3° ValiderLignes, 4° ValiderClient,... ou l'ordre inverse)

- Ordre par volume et complexité de messages échangés

- Ordre par complexité des règles de traitement des fonctions

- Ordre d'exécution des fonctions de la phase (selon le schéma de la dynamique)

- Ordre quelconque (ordre de spécification)

Le programmeur jugera du critère à utiliser qui correspondra le mieux à la situation donnée et qui lui permettra d'économiser au maximum ses efforts de développement.

4. Un point plus spécifique est à éclaircir, celui qui concerne la vérification des préconditions de chaque procédure. Le problème est le suivant : on observe souvent une redondance entre les préconditions de plusieurs procédures, et on désirerait ne pas devoir vérifier constamment les mêmes préconditions.

Face à cela, trois alternatives sont envisageables :

- Aucune vérification de préconditions ne sera effectuée car nous faisons une confiance absolue à la personne ayant élaboré les spécifications fonctionnelles et nous supposons que les messages (et donc les paramètres) sont acquis dans un état correct. Cette solution valable est évidemment une solution de facilité mais elle présente quelques dangers étant donné la possible complexité des transformations effectuées sur ces spécifications fonctionnelles.

- La solution opposée consiste à vérifier à chaque exécution d'une procédure la totalité de ses préconditions (à l'aide d'une nouvelle procédure de vérification créée à cet effet par exemple). Cette alternative est évidemment très lourde à supporter.

- Il faut trouver un mécanisme qui permette de ne pas devoir tester à nouveau une précondition déjà testée pour une autre procédure. Le problème est que même si 2 préconditions appartenant à des procédures différentes sont strictement identiques, rien ne prouve qu'entre les 2 moments de leur vérification, leurs éléments n'aient pas changé de valeur suite à l'exécution d'un quelconque traitement. Une précondition vraie à un certain moment peut toujours voir sa validité annulée par l'exécution des traitements d'une autre procédure.

Concrètement, ce mécanisme serait représenté par un tableau constamment mis à jour (par le module coordinateur par exemple) qui indiquerait dans sa première ligne quelle précondition a été testée ou doit être retestée. Nous trouverions dans les lignes suivantes de ce tableau les objets sur lesquels portent les différentes préconditions, et les colonnes représenteraient les différentes préconditions des procédures de l'application. Chaque exécution de procédure marquerait alors les cases des objets modifiés par les traitements et seules les préconditions ne possédant aucune case marquée devront être revérifiées.

Cependant, ce mécanisme ne procurerait pas d'avantages étant donné qu'il faudrait gérer de toute façon ce tableau à chaque appel de procédure (même quand celle-ci ne possède pas elle-même de préconditions), ce qui est assez complexe.

Dès lors, à moins de faire complètement confiance aux spécifications fonctionnelles, il est préférable, dans la situation actuelle, de tester toutes les préconditions d'une procédure lors de chaque appel à celle-ci. Un module auxiliaire spécifique à cette tâche et contenant toutes les procédures de vérification pourrait être construit et appelé par le module coordinateur en même temps que la procédure concernée.

VI.2.2 *Vue "Application - Machine-Application"*

A un certain moment, nous avons considéré la possibilité d'une "Machine-application" correspondant à une application IDA. Sans savoir si cette machine serait un module informatique ou non, plusieurs constatations nous ont conduit à abandonner cette idée :

- la découpe en fonctions est fondamentale et est à la base du processus de dérivation
- une application est une unité de planning quasi-autonome des autres applications et dont le cycle de développement doit être considéré séparément
- on trouve beaucoup moins d'applications dans un projet informatique que de phases et de fonctions, et donc de relations possibles entre elles

La création d'une machine-application fut évoquée car on désirait obtenir une vue collective de l'application informatique, permettant de la sorte une mise en commun plus forte des concepts similaires (même si une consolidation fut faîte au niveau des spécifications fonctionnelles). Cette vue collective remplacerait ou complèterait alors l'utilisation de modules auxiliaires.

Cette machine-application serait aussi vue comme un module coordinateur de niveau supérieur qui décrirait l'enchaînement des machines-phase mais cela peut être pleinement assuré par le module coordinateur défini auparavant.

A ce stade-ci, l'opportunité d'une machine-application ne semble donc présenter que peu d'intérêt. La figure VI.1 présente ce concept.

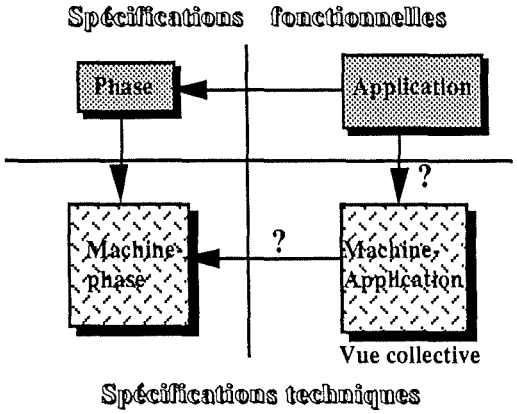


Figure VI.1
Vue Application - Machine-application

VI.2.3 *Changements de représentation de la base de données*

Un autre problème concerne l'application de la technique de changement de représentation de la mémoire des machines-phase.

En effet, les programmeurs se divisent souvent en 2 groupes ayant chacun leur opinion bien fondée à ce sujet :

- certains prétendent que les changements de représentation ne doivent se faire qu'au niveau du module de gestion de base de données **car**
  * c'est la tâche du spécialiste des bases de données qui en général s'occupera de ce module auxiliaire
  * la cohérence sera mieux assurée avec la représentation de la mémoire de chaque machine-phase (le module auxiliaire fait office de "couche de protection" en offrant des services sans dévoiler la façon dont ils sont réalisés)

* dans ce module, on est plus proche du niveau physique et les changements de représentation sont plus justifiés

- d'autres affirment que les changements de représentation de mémoire doivent être possibles dans tous les modules informatiques **car**

  * chaque machine-phase peut avoir une vue quelque peu différente de la base de données selon ses spécificités et ses besoins de traitements
  * les traitements sont liés aux données donc il faut que les changements se fassent au niveau de chaque module informatique afin de répercuter tous les changements de représentation sur les traitements concernés
  * le but de cette technique est avant tout d'aider le programmeur à essayer plusieurs alternatives possibles dans la conception des traitements et des données

Bien que cela reste discutable, l'application de la technique de changement de représentation doit être possible à l'intérieur de tous les modules informatiques car le programmeur doit pouvoir considérer plusieurs alternatives pour effectuer le meilleur choix.

Mais il faut que ces changements de représentation dispersés soient cohérents entre eux et respectent la cohérence et l'intégrité de la base de données globale de l'application informatique ce qui revient à dire qu'en fait peu de changements sont permis.

VI.2.4 *Critique du principe de la mémoire d'exécution*

Les techniques visant à introduire une mémoire d'exécution (c'est à dire une mémoire temporaire) au sein de la machine-phase sont criticables, et ce malgré les avantages que nous avons évoqués auparavant. Citons les 2 reproches les plus courants concernant l'application de telles techniques :

- La simplification attendue des structures de données échangées se justifierait surtout au niveau des gains de performance physique, ce qui n'est pas encore prouvé et de toute façon cela ne constitue pas vraiment notre préoccupation principale actuellement.

- Les procédures dérivées des fonctions perdent leur caractère continu et leur clarté car de nouvelles procédures et de nouveaux traitements doivent être définis. De plus, leur caractère réutilisable se réduit aussi fortement (selon le moment d'utilisation, on peut ne pas vouloir utiliser les "extensions" apportées par ces techniques).
  Donc, sémantiquement, les transformations engendrées par ces techniques d'introduction de mémoire auxiliaire sont douteuses. La correspondance Fonction - Procédure n'est plus vérifiée et on pourrait même affirmer, un peu abusivement certes, que les spécifications fonctionnelles sont violées.

Une autre idée fut soulevée : pourquoi ne pas considérer une mémoire d'exécution, globale à l'application informatique, contenant la plupart des paramètres spécifiés par la transformation de base des spécifications fonctionnelles ? Etant donné que le module coordinateur doit de toute façon gérer une mémoire qui lui est propre, pourquoi ne pas associer celle-ci à cette mémoire d'exécution (Figure VI.2) ?

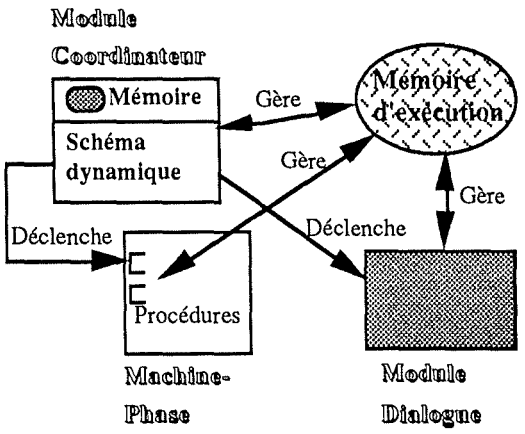


Figure VI.2
Mémoire d'exécution globale

Notons que la relation *Gère* signifie : consulte et met à jour.

L'idée est bien sûr tentante. Cela permettrait de stocker tous les paramètres dans une zone particulière et d'y avoir accès facilement, de n'avoir qu'une seule définition de ces données et d'assurer ainsi une communication aisée entre tous les composants de l'application informatique. Mais les inconvénients sont aussi nombreux : en plus de ceux qui viennent d'être cités, nous avons notamment le fait que les traitements des différents composants de l'application se complexifieront à cause de la gestion nécessaire de cette mémoire d'exécution (accès et mise à jour).
Néanmoins, cette proposition reste intéressante.

VI.2.5 *Contexte organisationnel*

Il ne faut jamais oublier que le but du processus de dérivation proposé n'est pas seulement d'ordre technique. Il faut en effet toujours prendre en compte l'aspect socio-organisationnel qui, déjà important dans tout le cycle de développement des systèmes d'information, est aussi primordial dans le cadre plus spécifique de cette dérivation.

Les objectifs poursuivis sont :

- Compréhension aisée des traitements de l'application par tous les membres concernés de l'entreprise
- Utilisation possible par tous des éléments développés
- Séparation coordonnée du travail
- Amélioration générale de la communication

D'un autre côté, il ne faut pas omettre non plus le fait que les choix établis dans le processus de dérivation ne sont pas seulement dictés par des impératifs techniques, mais qu'ils sont aussi établis par l'environnement organisationnel ainsi que par la culture de l'entreprise et de ses membres (dont les concepteurs de l'application).

Ces quelques remarques ne prétendent évidemment pas couvrir l'étendue des problèmes que l'on peut rencontrer ou fournir une solution définitive, elles évoquent seulement une gamme d'aspects auxquels il faut attacher une certaine importance.

CONCLUSION

L'intérêt, la rigueur et la nécessité des spécifications fonctionnelles dans le développement des systèmes d'information ont conduit assez naturellement à envisager des spécifications directement exécutables.

Dans le cadre du processus de dérivation proposé dans ce travail, le caractère exécutable des spécifications s'est traduit principalement par une continuité dans le développement et un automatisme maximal dans la dérivation des applications informatiques à partir de leurs spécifications.

Un ensemble de modèles et techniques de transformation des spécifications fonctionnelles fut défini et une proposition de structuration de tous ces concepts fut élaborée.

Se basant sur les concepts de module informatique, de techniques de transformation de signature et de techniques de raffinement de module proposés par Yves Pigneur pour élaborer les spécifications techniques d'une application, ce mémoire a permis d'expliquer et d'étendre les notions sous-jacentes à ceux-ci, de présenter un langage de spécification mais aussi de proposer une démarche générale de dérivation des applications informatiques et de tester l'application de tous les concepts étudiés dans 2 environnements différents. En outre, ce travail a également voulu mettre en évidence les problèmes principaux qui se sont posés et dont il faudra tenir compte dans les recherches futures.

Malheureusement, la théorie proposée est restreinte à un cadre sémantique bien défini (la méthode IDA) et elle ne représente bien sûr qu'une des multiples possibilités étudiées afin de concevoir et réaliser le plus automatiquement possible les applications informatiques.

Ce processus de dérivation, bien que possédant à ce stade-ci des limites d'application assez évidentes (nous avons notamment pu le constater lors de l'analyse de l'environnement IDA), représente quand même une aide plus qu'efficace pour le programmeur dans sa tâche de développement et de réalisation des applications informatiques. Celui-ci pourra disposer de tout un ensemble de moyens et d'outils réduisant considérablement les problèmes inhérents à son travail. Ces moyens permettront de faciliter et d'automatiser la construction des traitements de l'application et ils amélioreront le développement de son architecture globale, et ce dans la majorité des environnements physiques existants.

Il faut bien se rendre compte qu'une méthode "miracle" de réalisation automatique des applications informatiques ne peut être trouvée dans l'immensité de la "jungle" informatique actuelle. Les technologies évoluent constamment et les générations futures de matériel, de logiciel ou de méthodologies de développement apporteront leur part de nouveauté dans la recherche permanente de simplification et de systématisation lors de la réalisation des applications.

Cette évolution exige des adaptations suivies et une mise à jour fréquente, si ce n'est une remise en question, des théories établies.

Une solution générale est impossible à trouver et des contraintes conceptuelles ou physiques sont toujours présentes pour nous le rappeler.

Les travaux futurs devraient tenir compte de cet état de fait et essayer de proposer un processus de dérivation le plus général mais aussi le plus précis possible. Il faudra dès lors étudier diverses voies tant au niveau des spécifications, qu'à l'autre bout de la chaîne, c'est à dire au niveau des environnements physiques. Plusieurs possibilités sont envisageables : une extension ou une révision des spécifications fonctionnelles IDA, l'utilisation de méta-modèles, la création de nouvelles techniques de transformations plus performantes ou toute autre amélioration adaptant de mieux en mieux la démarche de dérivation proposée aux réalités informatiques existantes .

Cela risque évidemment de prendre beaucoup de temps et demandera des efforts soutenus mais l'enjeu qui est posé mérite bien cela.

BIBLIOGRAPHIE

[BODART-PIGNEUR,89]
F. Bodart, Y. Pigneur, *Conception assistée des Systèmes d'information : Méthode - Modèles - Outils,* 2ème édition, Masson, 1989

[BODART,90]
F. Bodart, *Introduction à la conception technique par dérivation systématique et continue à partir des spécifications fonctionnelles,* Document de travail confidentiel, FUNDP, Institut d'Informatique, Mars 1990

[CLARINVAL,81]
A. Clarinval, *Comprendre, Connaître et Maîtriser le Cobol, Norme ANSI COBOL 1974,* Travaux de l'Institut d'Informatique n°6, Presses Universitaires de Namur, Première édition, 1981

[DIJKSTRA,68]
E.W. Dijkstra, *The structure of the T.H.E Multiprogramming system,* Communications of the ACM, vol. 11, no 5, May 1968, pp341 - 346

[HAINAUT,86]
J-L. Hainaut, *Conception assistée des applications informatiques, Tome 2, Conception de la base de données,* Masson, 1986

[PARNAS, 72a]
D.L. Parnas, *On a "Buzzword": Hierarchical structure,* pp 186 - 189

[PETOUD,88]
I. Petoud, Y. Pigneur, *Applications de gestion hautement intéractives : génération automatique de l'interface à partir des spécifications fonctionnelles,* Journées Interuniversitaires Genève - Grenoble - Lausanne, Corsier, 1988

[PIGNEUR,89]
Y. Pigneur, *Construction d'une application informatique à partir de la spécification d'une phase IDA,* Memorandum technique M-31, Institut d'Informatique et d'Organisation, HEC, Lausanne, Avril 1989

[PIGNEUR,90]
Y. Pigneur, G. Maksay, *La dérivation de la machine logicielle d'une application intéractïve,* Séminaire Genève, Juillet 90

[ROESNER,85]
W. Roesner, *DESPATH : An ER manipulation language,* IEEE, Septembre 85, pp 72 - 81

[SOMMERVILLE,89]
I. Sommerville, *Software Engineering,* International Computer series, 3rd edition, Addison-Wesley, Wokingham, 1989

[-,86a]
DELTA Version 2.4, *Manuel de référence DELTA (COBOL),* DELTA SOFTWARE TECHNOLOGIE AG, Delta Manual n° MA 309, Janvier 1986

[-,86b]
DELTA Version 2.5, *On-line programming with DELTA,* DELTA SOFTWARE TECHNOLOGIE AG, Delta Manual n° MA 309.2, Octobre 1988

```
      ******************************************************************
      *         PROGRAMME DE TEST DE LA MACHINE-PHASE                  *
      ******************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CLI2.
       AUTHOR. Ph RUTTENS
       DATE-WRITTEN. Decembre 89
       DATE-COMPILED. Janvier 89
       ENVIRONMENT DIVISION.
       DATA DIVISION.

      ******************************************************************
        WORKING-STORAGE SECTION.
      ******************************************************************
      ******************************************************************
      **  DEFINITION DE L'ESPACE DE STOCKAGE POUR LES VARIABLES *
      **  DE DIALOGUE                                           *
      ******************************************************************

          01 OPTION PIC X VALUE SPACE.
          01 NOUVNOM PIC X(30) VALUE SPACES.
          01 NOUVLOC PIC X(30) VALUE SPACES.
          01 NRECH PIC 9(5) VALUE 0.
          01 MLOC PIC X(30) VALUE SPACES.
          01 INOM PIC X(30) VALUE SPACES.
          01 ILOC PIC X(30) VALUE SPACES.
          01 ICAT PIC X VALUE SPACE.
          01 LNUM PIC 9(5) VALUE 0.
          01 LNOM PIC X(30) VALUE SPACES.
          01 LLOC PIC X(30) VALUE SPACES.
          01 MSG PIC X(8) VALUE SPACES.

      ******************************************************************
      *     VARIABLES DE CONTROLE DE BOUCLE                            *
      ******************************************************************

          01 BTEST PIC X VALUE "0".
          01 ETAT PIC X VALUE "1".

      ******************************************************************
      *  DEFINITION DES LONGUEURS POUR LES VARIABLES DE                *
      *  DIALOGUE                                                      *
      ******************************************************************

          01 LOPTION     PIC 9(6) VALUE 1  COMP.
          01 LNOUVNOM    PIC 9(6) VALUE 30 COMP.
          01 LNOUVLOC    PIC 9(6) VALUE 30 COMP.
          01 LNRECH      PIC 9(6) VALUE 5  COMP.
          01 LMLOC       PIC 9(6) VALUE 30 COMP.
          01 LINOM       PIC 9(6) VALUE 30 COMP.
          01 LILOC       PIC 9(6) VALUE 30 COMP.
          01 LICAT       PIC 9(6) VALUE 1  COMP.
          01 LLNUM       PIC 9(6) VALUE 5  COMP.
          01 LLNOM       PIC 9(6) VALUE 30 COMP.
          01 LLLOC PIC 9(6)       VALUE 30 COMP.
```

```
*****************************************************************
*  DEFINITION DES NOMS DE VARIABLE DE DIALOGUE POUR LES  *
*  APPELS AUX SERVICES DE DIALOGUE                       *
*****************************************************************

    01 NOPTION PIC X(8) VALUE "(OPTION)".
    01 NNOUVNOM PIC X(9) VALUE "(NOUVNOM)".
    01 NNOUVLOC PIC X(9) VALUE "(NOUVLOC)".
    01 NNRECH PIC X(8) VALUE "(NRECH)".
    01 NMLOC PIC X(6) VALUE "(MLOC)".
    01 NINOM PIC X(6) VALUE "(INOM)".
    01 NILOC PIC X(6) VALUE "(ILOC)".
    01 NICAT PIC X(6) VALUE "(ICAT)".
    01 NLNUM PIC X(6) VALUE "(LNUM)".
    01 NLNOM PIC X(6) VALUE "(LNOM)".
    01 NLLOC PIC X(6) VALUE "(LLOC)".

*****************************************************************
*       LISTE DES ECRANS                                        *
*****************************************************************

    01 PRINCIP PIC X(8) VALUE "PRINCIP ".
    01 ECRAN1 PIC X(8) VALUE "ECRAN1   ".
    01 ECRAN2 PIC X(8) VALUE "ECRAN2   ".
    01 ECRAN3 PIC X(8) VALUE "ECRAN3   ".
    01 ECRAN4 PIC X(8) VALUE "ECRAN4   ".
    01 ECRAN5 PIC X(8) VALUE "ECRAN5   ".

    01 CLSELTBL PIC X(8) VALUE "CLSELTBL".
    01 TABVARS PIC X(21) VALUE "(ICAT LNUM LNOM LLOC)".
    01 CHAR PIC X(8) VALUE "CHAR     ".
    01 DW PIC X(8) VALUE "WRITE    ".

*****************************************************************
*  DEFINITION DES TYPES DE SERVICES POUR LES APPELS
*  A ISPF                                                       *
*****************************************************************

    01 DISPLAYE    PIC X(8) VALUE "DISPLAY ".
    01 LOG         PIC X(8) VALUE "LOG     ".
    01 TBADD       PIC X(8) VALUE "TBADD   ".
    01 TBCLOSE     PIC X(8) VALUE "TBCLOSE ".
    01 TBCREATE    PIC X(8) VALUE "TBCREATE".
    01 TBOPEN      PIC X(8) VALUE "TBOPEN  ".
    01 VDEFINE     PIC X(8) VALUE "VDEFINE ".
    01 VRESET      PIC X(8) VALUE "VRESET  ".
    01 TBERASE     PIC X(8) VALUE "TBERASE ".
    01 TBDELETE    PIC X(8) VALUE "TBDELETE".
    01 TBSKIP      PIC X(8) VALUE "TBSKIP  ".
    01 TBBOTTOM    PIC X(8) VALUE "TBBOTTOM".
    01 TBDISPL     PIC X(8) VALUE "TBDISPL ".
    01 TBTOP       PIC X(8) VALUE "TBTOP   ".
```

```
      ************************************************************
      *  PARAMETRES UTILISES POUR L'APPEL A LA MACHINE-PHASE **
      ************************************************************
          01 SPECIF PIC X(65).
          01 W-ENRCLI REDEFINES SPECIF.
             05 W-NOMC PIC X(30).
             05 W-LOCC PIC X(30).
             05 W-NUMC PIC 9(5).
          01 W-MODIFCLI REDEFINES SPECIF.
             05 W-NUMC8 PIC 9(5).
             05 W-LOCC2 PIC X(30).
             05 FILLER PIC X(30).
          01 W-SUPPRICLI REDEFINES SPECIF.
             05 W-NUMC2 PIC 9(5).
             05 FILLER PIC X(60).
          01 W-VALIDCLI REDEFINES SPECIF.
             05 W-NUMC3 PIC 9(5).
             05 W-CVAL PIC X.
             05 FILLER PIC X(59).
          01 W-NOMDE REDEFINES SPECIF.
             05 W-NUMC4 PIC 9(5).
             05 W-NO PIC X(30).
             05 FILLER PIC X(30).
          01 W-LOCDE REDEFINES SPECIF.
             05 W-NUMC5 PIC 9(5).
             05 W-LO PIC X(30).
             05 FILLER PIC X(30).
          01 W--CATDE REDEFINES SPECIF.
             05 W-NUMC6 PIC 9(5).
             05 W-CATC PIC X.
             05 FILLER PIC X(59).
          01 W-INITITER REDEFINES SPECIF.
             05 W-CATC2 PIC X.
             05 FILLER PIC X(64).
          01 W-TERMITER REDEFINES SPECIF.
             05 W-TEST PIC X.
             05 W-ITERM PIC X.
             05 FILLER PIC X(63).
          01 W-SUIVITER REDEFINES SPECIF.
             05 W-NUMC10 PIC 9(5).
             05 W-RESULT PIC X.
             05 FILLER PIC X(59).
          01 W-INSTMA REDEFINES SPECIF.
             05 W-OK PIC X.
             05 FILLER PIC X(64).
          01 W-BLANC REDEFINES SPECIF.
             05 FILLER PIC X(65).

          01 W-VIDE PIC X(30) VALUE SPACES.
          01 W-I3  PIC 9(5).
          01 W-I4  PIC X(30).

      ************************************************************
       PROCEDURE DIVISION.
```

```
*****************************************************************

   P1 SECTION.

   P1-PROG-GENERAL.
      CALL "M1INSTMA" USING BY REFERENCE W-INSTMA.
      IF W-OK = "F" PERFORM P1-FIN.
      CALL "ISPLINK" USING VDEFINE NOPTION OPTION CHAR LOPTIO
      CALL "ISPLINK" USING VDEFINE NNOUVNOM NOUVNOM CHAR LNOU
      CALL "ISPLINK" USING VDEFINE NNOUVLOC NOUVLOC CHAR LNOU
      CALL "ISPLINK" USING VDEFINE NNRECH NRECH CHAR LNRECH.
      CALL "ISPLINK" USING VDEFINE NMLOC MLOC CHAR LMLOC.
      CALL "ISPLINK" USING VDEFINE NINOM INOM CHAR LINOM.
      CALL "ISPLINK" USING VDEFINE NILOC ILOC CHAR LILOC.
      CALL "ISPLINK" USING VDEFINE NICAT ICAT CHAR LICAT.
      CALL "ISPLINK" USING VDEFINE NLNUM LNUM CHAR LLNUM.
      CALL "ISPLINK" USING VDEFINE NLNOM LNOM CHAR LLNOM.
      CALL "ISPLINK" USING VDEFINE NLLOC LLOC CHAR LLLOC.
      MOVE SPACES TO MSG.
      CALL "ISPLINK" USING TBOPEN CLSELTBL.
      IF RETURN-CODE NOT = 0
      CALL "ISPLINK" USING TBCREATE CLSELTBL TABVARS.
      PERFORM P3-SAISIR-OPTION.
      PERFORM P2-CORPS-BOUCLE UNTIL OPTION = 0.
      CALL "M1DESMAC" USING BY REFERENCE W-BLANC.
      PERFORM P1-FIN.

P1-FIN.
         CALL "ISPLINK" USING TBCLOSE CLSELTBL.
         CALL "ISPLINK" USING TBERASE CLSELTBL.
         CALL "ISPLINK" USING VRESET.
         STOP RUN.

*****************************************************************
 P2 SECTION.

  P2-CORPS-BOUCLE.
       IF OPTION = 1 PERFORM P3-ENRCLI.
       IF OPTION = 2 PERFORM P3-MAJCLI.
       IF OPTION = 3 PERFORM P3-MAJCLI.
       IF OPTION = 4 PERFORM P2-SELCLIENTSCAT.
       PERFORM P3-SAISIR-OPTION.

 P2-SELCLIENTSCAT.
       MOVE SPACES TO MSG.
       CALL "ISPLINK" USING DISPLAYE ECRAN4 MSG.
       IF RETURN-CODE NOT = 8 PERFORM P2-SELCLI2.

 P2-SELCLI2.
       CALL "ISPLINK" USING TBBOTTOM CLSELTBL.
       PERFORM P2-EFFAC UNTIL RETURN-CODE NOT = 0.
       MOVE ICAT TO W-CATC2.
       CALL "M1INITIT" USING BY REFERENCE W-INITITER.
       MOVE "F" TO W-ITERM.
       PERFORM P2-CORPS-LISTE UNTIL W-ITERM = "T".
```

```
           CALL "ISPLINK" USING TBTOP CLSELTBL.
           MOVE SPACES TO MSG.
           CALL "ISPLINK" USING TBDISPL CLSELTBL ECRAN5.

       P2-EFFAC.
           CALL "ISPLINK" USING TBDELETE CLSELTBL.

       P2-CORPS-LISTE.
           CALL "M1SUIVIT" USING BY REFERENCE W-SUIVITER.
           MOVE W-NUMC10 TO LNUM.
           MOVE W-I3 TO W-NUMC4.
           MOVE W-I3 TO W-NUMC5.
           MOVE W-RESULT TO W-TEST.
           CALL "M1TERMIT" USING BY REFERENCE W-TERMITER.
           IF W-ITERM = "F" PERFORM P2-CORPS2.

       P2-CORPS2.
           MOVE W-I3 TO W-NUMC4.
           MOVE W-I3 TO W-NUMC5.
           CALL "M1NOMDE_" USING BY REFERENCE W-NOMDE.
           MOVE W-NO TO W-I4.
           MOVE W-I4 TO LNOM.
           CALL "M1LOCDE_" USING BY REFERENCE W-LOCDE.
           MOVE W-LO TO LLOC.
           CALL "ISPLINK" USING TBADD CLSELTBL.

      ************************************************************
       P3 SECTION.

       P3-SAISIR-OPTION.
           CALL "ISPLINK" USING DISPLAYE PRINCIP MSG.
           IF RETURN-CODE = 8 MOVE 0 TO OPTION
                              ELSE MOVE SPACES TO MSG.


       P3-ENRCLI.
           MOVE SPACES TO MSG.
           CALL "ISPLINK" USING DISPLAYE ECRAN1 MSG.
           MOVE NOUVNOM TO W-NOMC.
           MOVE NOUVLOC TO W-LOCC.
           IF RETURN-CODE NOT = 8
                 CALL "M1ENRCLI" USING BY REFERENCE W-ENRCLI
                 MOVE "CLI007" TO MSG.

       P3-MAJCLI.
           MOVE "0" TO BTEST.
           MOVE SPACES TO MSG.
           PERFORM P3-M2 UNTIL BTEST = "1".

       P3-M2.
           CALL "ISPLINK" USING DISPLAYE ECRAN2 MSG.
           IF RETURN-CODE NOT = 8 PERFORM P3-M3
                             ELSE MOVE "1" TO BTEST
                                  MOVE SPACES TO MSG.
```

```
       P3-M3.
               MOVE NRECH TO W-NUMC3.
               CALL "M1VALIDC" USING BY REFERENCE W-VALIDCLI.
               IF W-CVAL = "T" PERFORM P4-MAJ2
                               MOVE "1" TO BTEST
                          ELSE MOVE "CLI009   " TO MSG.

      ***************************************************************
       P4 SECTION.

       P4-MAJ2.
                MOVE W-NUMC3 TO W-I3.
                MOVE W-I3 TO W-NUMC4.
                MOVE W-I3 TO W-NUMC5.
                CALL "M1NOMDE_" USING BY REFERENCE W-NOMDE.
                MOVE W-NO TO INOM.
                CALL "M1LOCDE_" USING BY REFERENCE W-LOCDE.
                MOVE W-LO TO ILOC.
                MOVE SPACES TO MSG.
              IF OPTION NOT = 3 CALL "ISPLINK" USING DISPLAYE ECRAN3
                IF RETURN-CODE NOT = 8 PERFORM P4-SUITMAJ2.

       P4-SUITMAJ2.
                MOVE MLOC TO W-LOCC2.
                IF OPTION = 2 PERFORM P4-MODIFLOCCLI
                              ELSE PERFORM P4-SUPPRICLI.

       P4-MODIFLOCCLI.
                MOVE W-I3 TO W-NUMC8.
                CALL "M1MODCLI" USING BY REFERENCE W-MODIFCLI.
                MOVE "CLI004   " TO MSG.

       P4-SUPPRICLI.
                MOVE W-I3 TO W-NUMC2.
                CALL "M1SUPPCL" USING BY REFERENCE W-SUPPRICLI.
                MOVE "CLI006   " TO MSG.

      ***************************************************************
```

FILE: PHASE COBOL A VM/IS 5.1

```
*****************************************************************
*   PROGRAMME REPRESENTANT LA MACHINE-PHASE CLIENT              *
*****************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. MACHPHAS.
 AUTHOR. Ph RUTTENS
 DATE-WRITTEN. Octobre 89
 DATE-COMPILED. Novembre 89
 ENVIRONMENT DIVISION.
 DATA DIVISION.

*****************************************************************
 WORKING-STORAGE SECTION.
*****************************************************************

    77 PGM-NOM PIC X(8).
    01 W-INTERM PIC X(30).
    01 W-INTERM2 PIC 9(5).
*****************************************************************
*  PARAMETRES UTILISES POUR L'APPEL AU MODULE GESTION/BD        *
*****************************************************************
    01 BDPARAM PIC X(65).
    01 W-VALIDLOC REDEFINES BDPARAM.
       05 W-LOCC PIC X(30).
       05 W-PRESENT PIC X.
       05 FILLER PIC X(34).
    01 W-REFLOC REDEFINES BDPARAM.
       05 W-LOCC2 PIC X(30).
       05 W-L PIC 9(5).
       05 FILLER PIC X(30).
    01 W-MODLOC REDEFINES BDPARAM.
       05 W-NUMC PIC 9(5).
       05 W-L2 PIC 9(5).
       05 FILLER PIC X(55).
    01 W-SUPPCLI REDEFINES BDPARAM.
       05 W-NUMC2 PIC 9(5).
       05 FILLER PIC X(60).
    01 W-VALCLI REDEFINES BDPARAM.
       05 W-NUMC3 PIC 9(5).
       05 W-CVAL PIC X.
       05 FILLER PIC X(59).
    01 W-NOMCLI REDEFINES BDPARAM.
       05 W-NUMC4 PIC 9(5).
       05 W-NOMC PIC X(30).
       05 FILLER PIC X(30).
    01 W-ENREGCLI REDEFINES BDPARAM.
       05 W-NOMC2 PIC X(30).
       05 W-L3 PIC 9(5).
       05 W-NUMC5 PIC 9(5).
       05 FILLER PIC X(25).
    01 W-CHERCHLOC REDEFINES BDPARAM.
       05 W-L4 PIC 9(5).
       05 W-LOCC3 PIC X(30).
       05 FILLER PIC X(30).
    01 W-CATCLI REDEFINES BDPARAM.
```

```
           05 W-NUMC6 PIC 9(5).
           05 W-CATC PIC X.
           05 FILLER PIC X(59).
       01 W-CATSEL REDEFINES BDPARAM.
           05 W-CATC2 PIC X.
           05 FILLER PIC X(64).
       01 W-SUIVIT REDEFINES BDPARAM.
           05 W-NUMC9 PIC 9(5).
           05 W-RESULT PIC X.
           05 FILLER PIC X(59).
       01 W-TERMIT REDEFINES BDPARAM.
           05 W-TEST PIC X.
           05 W-ITERM PIC X.
           05 FILLER PIC X(63).
       01 W-INSTMA REDEFINES BDPARAM.
           05 BDSTATUT PIC X.
           05 FILLER PIC X(64).
       01 W-BLANC REDEFINES BDPARAM.
           05 FILLER PIC X(65).

*****************************************************************
 LINKAGE SECTION.
*****************************************************************

*****************************************************************
*     PARAMETRES DES PRIMITIVES DE LA MACHINE-PHASE             *
*****************************************************************
       01 ARGUMENT PIC X(65).
       01 L-ENRCLI REDEFINES ARGUMENT.
           05 L-NOMC PIC X(30).
           05 L-LOCC PIC X(30).
           05 L-NUMC PIC 9(5).
       01 L-MODIFCLI REDEFINES ARGUMENT.
           05 L-NUMC8 PIC 9(5).
           05 L-LOCC2 PIC X(30).
           05 FILLER PIC X(30).
       01 L-SUPPRICLI REDEFINES ARGUMENT.
           05 L-NUMC2 PIC 9(5).
           05 FILLER PIC X(60).
       01 L-VALIDCLI REDEFINES ARGUMENT.
           05 L-NUMC3 PIC 9(5).
           05 L-CVAL PIC X.
           05 FILLER PIC X(59).
       01 L-NOMDE REDEFINES ARGUMENT.
           05 L-NUMC4 PIC 9(5).
           05 L-NOMC2 PIC X(30).
           05 FILLER PIC X(30).
       01 L-LOCDE REDEFINES ARGUMENT.
           05 L-NUMC5 PIC 9(5).
           05 L-LOCC3 PIC X(30).
           05 FILLER PIC X(30).
       01 L-CATDE REDEFINES ARGUMENT.
           05 L-NUMC6 PIC 9(5).
           05 L-CATC PIC X.
           05 FILLER PIC X(59).
```

```
       01 L-INITITER REDEFINES ARGUMENT.
          05 L-CATC2 PIC X.
          05 FILLER PIC X(64).
       01 L-TERMITER REDEFINES ARGUMENT.
          05 L-TEST PIC X.
          05 L-ITERM PIC X.
          05 FILLER PIC X(63).
       01 L-SUIVITER REDEFINES ARGUMENT.
          05 L-NUMC7 PIC 9(5).
          05 L-RES PIC X.
          05 FILLER PIC X(59).
       01 L-INSTMA REDEFINES ARGUMENT.
          05 L-OK PIC X.
          05 FILLER PIC X(64).
       01 L-BLANC REDEFINES ARGUMENT.
          05 FILLER PIC X(65).


      ************************************************************
       PROCEDURE DIVISION USING ARGUMENT.
      ************************************************************

           ENTRY "M1INSTMA" USING L-INSTMA.
      *      Procedure INSTALLER : OK                            *
           MOVE "OUVRIRBD" TO PGM-NOM.
           CALL PGM-NOM USING BY REFERENCE W-INSTMA.
           CANCEL PGM-NOM.
           IF BDSTATUT = "C" MOVE "T" TO L-OK
                             ELSE MOVE "F" TO L-OK.
           GOBACK.



           ENTRY "M1ENRCLI" USING L-ENRCLI.
      *     Procedure ENREGISTRERCLIENT (NomC,LocC) : NumC     *
           MOVE L-LOCC TO W-INTERM.
           MOVE W-INTERM TO W-LOCC.
           MOVE W-INTERM TO W-LOCC2.
           MOVE "VALIDLOC" TO PGM-NOM.
           CALL PGM-NOM USING BY REFERENCE W-VALIDLOC.
           CANCEL PGM-NOM.
           IF W-PRESENT = "T" MOVE "REFERLOC" TO PGM-NOM
                              ELSE MOVE "INSERERL" TO PGM-NOM.
           CALL PGM-NOM USING BY REFERENCE W-REFLOC.
           CANCEL PGM-NOM.
           MOVE L-NOMC TO W-NOMC2.
           MOVE W-L TO W-INTERM2.
           MOVE W-INTERM2 TO W-L3.
           MOVE "ENREGCLI" TO PGM-NOM.
           CALL PGM-NOM USING BY REFERENCE W-ENREGCLI.
           CANCEL PGM-NOM.
           MOVE W-NUMC5 TO L-NUMC.
           GOBACK.
```

FILE: PHASE COBOL A VM/IS 5.1

```
       ENTRY "M1MODCLI" USING L-MODIFCLI.
      *        Procedure MODIFIERLOCALITECLIENT (NumC,LocC)    *
       MOVE L-LOCC2 TO W-LOCC.
       MOVE "VALIDLOC" TO PGM-NOM.
       CALL PGM-NOM USING BY REFERENCE W-VALIDLOC.
       CANCEL PGM-NOM.
       IF W-PRESENT = "T" MOVE "REFERLOC" TO PGM-NOM
                          ELSE MOVE "INSERERL" TO PGM-NOM.
       CALL PGM-NOM USING BY REFERENCE W-REFLOC.
       CANCEL PGM-NOM.
       MOVE L-NUMC8 TO W-NUMC.
       MOVE W-L TO W-L2.
       MOVE "MODIFLOC" TO PGM-NOM.
       CALL PGM-NOM USING BY REFERENCE W-MODLOC.
       CANCEL PGM-NOM.
       GOBACK.



       ENTRY "M1SUPPCL" USING L-SUPPRICLI.
      *     Procedure SUPPRIMERCLIENT (NumC)                   *
       MOVE L-NUMC2 TO W-NUMC2.
       MOVE "SUPPCLI_" TO PGM-NOM.
       CALL PGM-NOM USING W-SUPPCLI.
       CANCEL PGM-NOM.
       GOBACK.



       ENTRY "M1VALIDC" USING L-VALIDCLI.
      *     Procedure VALIDERCLIENT (NumC) : Cval          *
       MOVE L-NUMC3 TO W-NUMC3.
       MOVE "F" TO W-CVAL.
       MOVE "VALIDCLI" TO PGM-NOM.
       CALL PGM-NOM USING BY REFERENCE W-VALCLI.
       CANCEL PGM-NOM.
       MOVE W-CVAL TO L-CVAL.
       GOBACK.



       ENTRY "M1NOMDE_" USING L-NOMDE.
      *       Procedure NOMDE (NumC) : NomC                  *
       MOVE L-NUMC4 TO W-NUMC4.
       MOVE "NOMCLI__" TO PGM-NOM.
       CALL PGM-NOM USING BY REFERENCE W-NOMCLI.
       CANCEL PGM-NOM.
       MOVE W-NOMC TO L-NOMC2.
       GOBACK.



       ENTRY "M1CATDE_" USING L-CATDE.
      *       Procedure CATEGORIEDE (NumC) : CatC            *
```

```
           MOVE L-NUMC6 TO W-NUMC6.
           MOVE "CATCLI__" TO PGM-NOM.
           CALL PGM-NOM USING BY REFERENCE W-CATCLI.
           CANCEL PGM-NOM.
           MOVE W-CATC TO L-CATC.
           GOBACK.


           ENTRY "M1LOCDE_" USING L-LOCDE.
      *     Procedure LOCALITEDE (NumC) : LocC                    *
           MOVE L-NUMC5 TO W-NUMC.
           MOVE "LOCCLI__" TO PGM-NOM.
           CALL PGM-NOM USING BY REFERENCE W-MODLOC.
           CANCEL PGM-NOM.
           MOVE W-L2 TO W-INTERM2.
           MOVE W-INTERM2 TO W-L4.
           MOVE "CHERCHLO"  TO PGM-NOM.
           CALL PGM-NOM USING BY REFERENCE W-CHERCHLOC.
           CANCEL PGM-NOM.
           MOVE W-LOCC3 TO L-LOCC3.
           GOBACK.


           ENTRY "M1INITIT" USING L-INITITER.
      *       Procedure INITITERATION (CatC)                      *
           MOVE L-CATC2 TO W-CATC2.
           MOVE "CATSEL__" TO PGM-NOM.
           CALL PGM-NOM USING BY REFERENCE W-CATSEL.
           CANCEL PGM-NOM.
           GOBACK.



           ENTRY "M1TERMIT" USING L-TERMITER.
      *       Procedure TERMITERATION :Iterm                      *
           MOVE L-TEST TO W-TEST.
           MOVE "TERMIT__" TO PGM-NOM.
           CALL PGM-NOM USING BY REFERENCE W-TERMIT.
           CANCEL PGM-NOM.
           MOVE W-ITERM TO L-ITERM.
           GOBACK.


           ENTRY "M1SUIVIT" USING L-SUIVITER.
      *       Procedure SUIVITERATION : NumC                      *
           MOVE "SUIVITER" TO PGM-NOM.
           CALL PGM-NOM USING BY REFERENCE W-SUIVIT.
           CANCEL PGM-NOM.
           MOVE W-NUMC9 TO L-NUMC7.
           MOVE W-RESULT TO L-RES.
           GOBACK.



           ENTRY "M1DESMAC" USING L-BLANC.
```

FILE: PHASE COBOL A VM/IS 5.1

```
*      Procedure DESINSTALLER                                          *
       MOVE "FERMERBD" TO PGM-NOM.
       CALL PGM-NOM USING BY REFERENCE W-BLANC.
       CANCEL PGM-NOM.
       GOBACK.

**********************************************************************
```

FILE: BTRANSFO MACRO A VM/IS 5.1

```
*****************************************************************
*    PROGRAMME D'ACCES ET DE GESTION DE LA BD                   *
*****************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. MODACCES.
 AUTHOR. Ph RUTTENS
 DATE-WRITTEN. Octobre 89
 DATE-COMPILED. Novembre 89
 ENVIRONMENT DIVISION.
 DATA DIVISION.

*****************************************************************
 WORKING-STORAGE SECTION.
*****************************************************************

*****************************************************************
*     VARIABLES UTILISEES PAR LES REQUETES SQL/DS **************
*****************************************************************
       EXEC SQL BEGIN DECLARE SECTION END-EXEC.

    77 TEMPLOC      PIC X(30).
    77 LOCC         PIC X(30).
    77 VALIDE       PIC X.
    77 L            PIC S9(5) COMP.
    77 NUMC         PIC S9(5) COMP.
    77 NOMC         PIC X(30).
    77 TEMPID       PIC S9(5) COMP.
    77 CATC         PIC X.
    77 ITERM        PIC X.
    77 BDSTATUT     PIC X.
    01 USERID       PIC X(8) VALUE 'DPI025  '.
    01 PASSW        PIC X(8) VALUE '25DPI0  '.
    01 V            PIC X VALUE 'V'.
    01 NOTFND       PIC S9(9) COMP VALUE +100.

       EXEC SQL END DECLARE SECTION END-EXEC.

       EXEC SQL INCLUDE SQLCA END-EXEC.

    01 W-TEST PIC X.

*****************************************************************
 LINKAGE SECTION.
*****************************************************************

*****************************************************************
*      LISTE DES PARAMETRES DES SERVICES OFFERTS  *
*****************************************************************
    01 L-BDPARAM PIC X(65).
    01 L-VALIDLOC REDEFINES L-BDPARAM.
       05 L-BD-LOCC PIC X(30).
       05 L-BD-PRESENT PIC X.
       05 FILLER PIC X(34).
    01 L-REFLOC REDEFINES L-BDPARAM.
       05 L-BD-LOCC2 PIC X(30).
```

```
           05 L-BD-L PIC 9(5).
           05 FILLER PIC X(30).
       01 L-MODLOC REDEFINES L-BDPARAM.
           05 L-BD-NUMC PIC 9(5).
           05 L-BD-L2 PIC 9(5).
           05 FILLER PIC X(55).
       01 L-SUPPCLI REDEFINES L-BDPARAM.
           05 L-BD-NUMC2 PIC 9(5).
           05 FILLER PIC X(60).
       01 L-VALCLI REDEFINES L-BDPARAM.
           05 L-BD-NUMC3 PIC 9(5).
           05 L-BD-CVAL PIC X.
           05 FILLER PIC X(59).
       01 L-NOMCLI REDEFINES L-BDPARAM.
           05 L-BD-NUMC4 PIC 9(5).
           05 L-BD-NOMC PIC X(30).
           05 FILLER PIC X(30).
       01 L-ENREGCLI REDEFINES L-BDPARAM.
           05 L-BD-NOMC2 PIC X(30).
           05 L-BD-L3 PIC 9(5).
           05 L-BD-NUMC5 PIC 9(5).
           05 FILLER PIC X(25).
       01 L-CHERCHLOC REDEFINES L-BDPARAM.
           05 L-BD-L4 PIC 9(5).
           05 L-BD-LOCC3 PIC X(30).
           05 FILLER PIC X(30).
       01 L-CATCLI REDEFINES L-BDPARAM.
           05 L-BD-NUMC6 PIC 9(5).
           05 L-BD-CATC PIC X.
           05 FILLER PIC X(59).
       01 L-CATSEL REDEFINES L-BDPARAM.
           05 L-BD-CATC2 PIC X.
           05 FILLER PIC X(64).
       01 L-TERMIT REDEFINES L-BDPARAM.
           05 L-BD-TEST PIC X.
           05 L-BD-ITERM PIC X.
           05 FILLER PIC X(63).
       01 L-SUIVIT REDEFINES L-BDPARAM.
           05 L-BD-NUMC9 PIC 9(5).
           05 L-BD-RES PIC X.
           05 FILLER PIC X(59).
       01 L-INSTMA REDEFINES L-BDPARAM.
           05 L-BD-BDSTATUT PIC X.
           05 FILLER PIC X(64).
       01 L-BLANC REDEFINES L-BDPARAM.
           05 FILLER PIC X(65).

*****************************************************************
 PROCEDURE DIVISION USING L-BDPARAM.
*****************************************************************

      ENTRY 'VALIDLOC' USING L-VALIDLOC.
*       Validation de la localite d'un client                   *
      MOVE L-BD-LOCC TO LOCC.
      EXEC SQL DECLARE C1 CURSOR FOR
```

```
                SELECT LOCALITE
                FROM LOCALITE
                WHERE LOCALITE = :LOCC
        END-EXEC.
        EXEC SQL OPEN C1 END-EXEC.
        EXEC SQL FETCH C1 INTO :TEMPLOC END-EXEC.
        IF SQLCODE = NOTFND   MOVE 'F' TO L-BD-PRESENT
                              ELSE MOVE 'T' TO L-BD-PRESENT.

        EXEC SQL CLOSE C1 END-EXEC.
        GOBACK.



        ENTRY 'REFERLOC' USING L-REFLOC.
      *       Recherche de l'identifiant d'une localite    *
        MOVE L-BD-LOCC2 TO LOCC.
        EXEC SQL DECLARE C2 CURSOR FOR
                SELECT IDLOCALITE
                FROM LOCALITE
                WHERE LOCALITE = :LOCC
        END-EXEC.
        EXEC SQL OPEN C2 END-EXEC.
        EXEC SQL FETCH C2 INTO :L END-EXEC.
        EXEC SQL CLOSE C2 END-EXEC.
        MOVE L TO L-BD-L.
        GOBACK.



        ENTRY 'INSERERL' USING L-REFLOC.
      *     Insertion d'une nouvelle localite               *
        MOVE L-BD-LOCC2 TO LOCC.
        EXEC SQL DECLARE C3 CURSOR FOR
                SELECT MAX(IDLOCALITE)+1
                FROM LOCALITE
        END-EXEC.
        EXEC SQL OPEN C3 END-EXEC.
        EXEC SQL FETCH C3 INTO :L END-EXEC.
        EXEC SQL CLOSE C3 END-EXEC.
        IF (L = 0) OR (SQLCODE = NOTFND) MOVE 1 TO L.
        EXEC SQL DECLARE C4 CURSOR FOR
                INSERT INTO
                LOCALITE (IDLOCALITE , LOCALITE)
                VALUES (:L , :LOCC)
        END-EXEC.
        EXEC SQL OPEN C4 END-EXEC.
        EXEC SQL PUT C4 END-EXEC.
        EXEC SQL CLOSE C4 END-EXEC.
        MOVE L TO L-BD-L.
        GOBACK.



        ENTRY 'ENREGCLI' USING L-ENREGCLI.
```

FILE: BTRANSFO MACRO A VM/IS 5.1

```
*       Insertion d'un nouveau client                          *
        MOVE L-BD-NOMC2 TO NOMC.
        MOVE L-BD-L3 TO L.
        MOVE 'N' TO CATC.
        EXEC SQL DECLARE C5 CURSOR FOR
                 SELECT MAX(NUMERO)+1
                 FROM CLIENT
        END-EXEC.
        EXEC SQL OPEN C5 END-EXEC.
        EXEC SQL FETCH C5 INTO :NUMC END-EXEC.
        EXEC SQL CLOSE C5 END-EXEC.
        IF (NUMC = 0) OR (SQLCODE = NOTFND)   MOVE 1 TO NUMC.
        EXEC SQL DECLARE C6 CURSOR FOR
                 INSERT INTO CLIENT
                        (NUMERO , NOM , IDLOCALITE , CATEGORIE)
                        VALUES (:NUMC , :NOMC , :L , :CATC)
        END-EXEC.
        EXEC SQL OPEN C6 END-EXEC.
        EXEC SQL PUT C6 END-EXEC.
        EXEC SQL CLOSE C6 END-EXEC.
        MOVE NUMC TO L-BD-NUMC5.
        GOBACK.



        ENTRY 'MODIFLOC' USING L-MODLOC.
*   Modification de l'identifiant de la localite d'un client :
        MOVE L-BD-NUMC TO NUMC.
        MOVE L-BD-L2 TO L.
        EXEC SQL DECLARE C7 CURSOR FOR
                 SELECT IDLOCALITE FROM CLIENT
                 WHERE NUMERO = :NUMC
                 FOR UPDATE OF IDLOCALITE
        END-EXEC.
        EXEC SQL OPEN C7 END-EXEC.
        EXEC SQL FETCH C7 INTO :TEMPID END-EXEC.
        EXEC SQL UPDATE CLIENT SET IDLOCALITE = :L
                 WHERE CURRENT OF C7 END-EXEC.
        EXEC SQL CLOSE C7 END-EXEC.
        GOBACK.



        ENTRY 'SUPPCLI_' USING L-SUPPCLI.
*         Suppression d'un client                              *
        MOVE L-BD-NUMC2 TO NUMC.
        EXEC SQL DECLARE C8 CURSOR FOR
                 SELECT IDLOCALITE FROM CLIENT
                 WHERE NUMERO = :NUMC
        END-EXEC.
        EXEC SQL OPEN C8 END-EXEC.
        EXEC SQL FETCH C8 INTO :TEMPID END-EXEC.
        EXEC SQL DELETE FROM CLIENT
                 WHERE CURRENT OF C8  END-EXEC.
        EXEC SQL CLOSE C8 END-EXEC.
```

```
           GOBACK.


           ENTRY 'VALIDCLI' USING L-VALCLI.
      *          Validation d'un client                          *
           MOVE L-BD-NUMC3 TO NUMC.
           EXEC SQL DECLARE C9 CURSOR FOR
                    SELECT IDLOCALITE
                    FROM CLIENT
                    WHERE NUMERO = :NUMC
           END-EXEC.
           EXEC SQL OPEN C9 END-EXEC.
           EXEC SQL FETCH C9 INTO :TEMPID END-EXEC.
           IF SQLCODE = NOTFND   MOVE 'F' TO L-BD-CVAL
                              ELSE MOVE 'T' TO L-BD-CVAL.
           EXEC SQL CLOSE C9 END-EXEC.
           GOBACK.


           ENTRY 'CATSEL___' USING L-CATSEL.
      *        Selection des clients d'une certaine categorie    *
           MOVE L-BD-CATC2 TO CATC.
           EXEC SQL DECLARE CLSELECTIONNES CURSOR FOR
                    SELECT NUMERO , NOM , IDLOCALITE , CATEGORIE
                    FROM CLIENT
                    WHERE CATEGORIE = :CATC
                    ORDER BY NUMERO
           END-EXEC.
           EXEC SQL OPEN CLSELECTIONNES END-EXEC.

           GOBACK.


           ENTRY 'TERMIT___' USING L-TERMIT.
      *          Test de presence des clients selectionnes       *
           IF L-BD-TEST = '0' MOVE 'T' TO L-BD-ITERM
                              ELSE MOVE 'F' TO L-BD-ITERM.
           GOBACK.


           ENTRY 'SUIVITER' USING L-SUIVIT.
      *             Lecture du client selectionne suivant        *
           MOVE 'N' TO L-BD-RES.
           EXEC SQL FETCH CLSELECTIONNES
                    INTO :NUMC, :NOMC, :L, :CATC
           END-EXEC.
           MOVE NUMC TO L-BD-NUMC9.
           IF SQLCODE = NOTFND  MOVE '0' TO L-BD-RES.
           GOBACK.
```

FILE: BTRANSFO MACRO A VM/IS 5.1

```
      ENTRY 'OUVRIRBD' USING L-INSTMA.
*       Connection a la base de donnees                    *
      MOVE 'C' TO L-BD-BDSTATUT.
      EXEC SQL WHENEVER SQLERROR GOTO FIN END-EXEC.
      EXEC SQL CONNECT :USERID IDENTIFIED BY :PASSW END-EXEC.
      EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
      GOBACK.

    FIN.
      MOVE 'N' TO L-BD-BDSTATUT.
      GOBACK.



      ENTRY 'NOMCLI__' USING L-NOMCLI.
*           Recherche du nom d'un client                  *
      MOVE L-BD-NUMC4 TO NUMC.
      EXEC SQL DECLARE C10 CURSOR FOR
               SELECT NOM
               FROM CLIENT
               WHERE NUMERO = :NUMC
      END-EXEC.
      EXEC SQL OPEN C10 END-EXEC.
      EXEC SQL FETCH C10 INTO :NOMC END-EXEC.
      EXEC SQL CLOSE C10 END-EXEC.
      MOVE NOMC TO L-BD-NOMC.
      GOBACK.



      ENTRY 'LOCCLI__' USING L-MODLOC.
*     Recherche de l'identifiant de la localite d'un client
      MOVE L-BD-NUMC TO NUMC.
      EXEC SQL DECLARE C11 CURSOR FOR
               SELECT IDLOCALITE
               FROM CLIENT
               WHERE NUMERO = :NUMC
      END-EXEC.
      EXEC SQL OPEN C11 END-EXEC.
      EXEC SQL FETCH C11 INTO :L END-EXEC.
      EXEC SQL CLOSE C11 END-EXEC.
      MOVE L TO L-BD-L2.
      GOBACK.



      ENTRY 'CHERCHLO' USING L-CHERCHLOC.
* Recherche du nom d'une localite a partir de son identifian
      MOVE L-BD-L4 TO L.
      EXEC SQL DECLARE C12 CURSOR FOR
               SELECT LOCALITE
               FROM LOCALITE
               WHERE IDLOCALITE = :L
      END-EXEC.
```

```
           EXEC SQL OPEN C12 END-EXEC.
           EXEC SQL FETCH C12 INTO :LOCC END-EXEC.
           EXEC SQL CLOSE C12 END-EXEC.
           MOVE LOCC TO L-BD-LOCC3.
           GOBACK.



           ENTRY 'CATCLI__' USING L-CATCLI.
      *      Recherche de la categorie d'un client                *
           MOVE L-BD-NUMC6 TO NUMC.
           EXEC SQL DECLARE C13 CURSOR FOR
                    SELECT CATEGORIE
                    FROM CLIENT
                    WHERE NUMERO = :NUMC
           END-EXEC.
           EXEC SQL OPEN C13 END-EXEC.
           EXEC SQL FETCH C13 INTO :CATC END-EXEC.
           EXEC SQL CLOSE C13 END-EXEC.
           MOVE CATC TO L-BD-CATC.
           GOBACK.



           ENTRY 'FERMERBD' USING L-BLANC.
      *  Fermeture des traitements portant sur la base de donnees *
           EXEC SQL CLOSE CLSELECTIONNES END-EXEC.
           EXEC SQL COMMIT WORK END-EXEC.

           GOBACK.

      ***************************************************************
```

```
FILE: PRINCIP  PANEL    A    VM/IS 5.1

%------------------------------ TRAITEMENT CLIENTS ------------------------------
+
% VOTRE CHOIX ===>_OPTION%
%
%      1 +. Enregistrer un nouveau client
%      2 +. Modifier la localite du client
%      3 +. Supprimer un client
%      4 +. Liste des clients d'une categorie
+
+Frappez la commande%END+pour terminer.
)INIT
&OPTION = ' '
 .CURSOR = OPTION

)PROC

  VER(&OPTION,LIST,1,2,3,4,MSG=CLI001)

)END
```

```
FILE: ECRAN1    PANEL     A    VM/IS 5.1

%------------------------------ ENREGISTREMENT CLIENT ------------------------------
+
%COMMAND ===>_ZCMD                                                              %
+
+
+
+    Entrez le nom du client:   _NOUVNOM                                        +
+
+    Entrez la localite du client: _NOUVLOC                                       +
+
+
+Frappez la commande%END+pour revenir au menu principal.
+
)INIT
&NOUVNOM = '                                        '
&NOUVLOC = '                                        '
  .CURSOR = NOUVNOM
)PROC
  VER (&NOUVNOM,NONBLANK)
  VER (&NOUVLOC,NONBLANK)
)END
```

```
FILE: ECRAN3    PANEL     A    VM/IS 5.1

%------------------------------ MODIFICATION LOCALITE ------------------------------
+
%COMMAND ====>_ZCMD                                                              %
+
+
+
+                    Nom du client:   _INOM                                  +
+
+                    Localite du client: _ILOC                                  +
+
+          Entrez la nouvelle localite : _MLOC                                   +

+
+Frappez la commande%END+pour revenir au menu precedent.
+
)INIT
&MLOC = '
  .CURSOR = MLOC
)PROC
  VER (&MLOC,NONBLANK)
)END
```

```
FILE: ECRAN4    PANEL     A    VM/IS 5.1

%------------------------------ CATEGORIE SELECTIONNEE ------------------------------
+
%COMMAND ===>_ZCMD
+
+
%          Entrez la categorie selectionnee: _ICAT+   %(N,R,A,D)+
+
+
+Frappez la commande%END+pour revenir au menu principal
+
+
)INIT
&ZCMD = ' '
&ICAT = ' '
  .CURSOR = ICAT
)PROC

  VER (&ICAT,NONBLANK,LIST,A,N,R,D,MSG=CLI005)

)END
```

```
FILE: ECRAN5   PANEL    A   VM/IS 5.1

)ATTR
  $ TYPE (OUTPUT) INTENS(HIGH)
)BODY
%------------------------- LISTE DES CLIENTS SELECTIONNES -------------------------
+
%COMMAND ===>_ZCMD
+
+      CAT    NUMERO   NOM                                  LOCALITE
+
)MODEL
_TSEL $ICAT $LNUM    $LNOM                                   $LLOC
)INIT
  &ZCMD=' '
  &TSEL = ' '
)PROC
  VER(&ZCMD,LIST,END,MSG=CLI001)
)END
```

```
FILE: CLI00     MESSAGE  A    VM/IS 5.1


CLI001 'CHOIX INVALIDE'            .ALARM=YES
'FRAPPEZ 1,2,3 OU 4...'

CLI002 'CLIENT INEXISTANT'         .ALARM=YES
'CE CLIENT N'EST PAS PRESENT DANS LA BASE DE DONNEES'

CLI003 'NUMERO INVALIDE'           .ALARM=YES
'ENTREZ UN NUMERO DE CLIENT NON-VIDE...'

CLI004 'LOCALITE MODIFIEE'
'LE CLIENT CHOISI POSSEDE UNE NOUVELLE LOCALITE'

CLI005 'CATEGORIE INVALIDE'        .ALARM=YES
'FRAPPEZ N,A,R OU D...'

CLI006 'CLIENT SUPPRIME'
'LE CLIENT CHOISI EST SUPPRIME DE LA BASE DE DONNEES'

CLI007 'CLIENT ENREGISTRE'
'LE CLIENT DECRIT EST ENREGISTRE DANS LA BASE DE DONNEES'

CLI009 'CLIENT INEXISTANT'
'LE CLIENT DECRIT N'EST PAS PRESENT DANS LA BD                '
```

ANNEXE B - APPLICATION DEVELOPPEE EN DELTA

```
**PDL*8909271515/FDCLIENT/00/
.SET-F9=NUMERO
.DEF-F4=RDCLIENT
.DEF-FN=CLIENT
.ADD  DCMIA
.SL  WORK01
     01   KEY-FI-START   PIC  X(3)   VALUE  LOW-VALUE.
     01   KEY-FI-END     PIC  X(3)   VALUE  HIGH-VALUE.
```

```
**PDL*8909271515/RDCLIENT/00/
.EXEC   FGEN
.  R-01
     05,-NUMERO,  9(5)
     05,-NOM,  X(30)
     05,-LOCALITE,X(30)
     05,-CATEGORIE,X
.END
```

```
**PDL*9007061003/MENUP/00/
.PROG-TP01,PROGMODE=TRANSACTION
.OSP,MENU,LAYOUT-DESCRIPTION=ECRANMC
.OSP-MENUP
.TITLE
                    MENU PRINCIPAL
                    ______________

.TEXT
              1 . ENREGISTRER NOUVEAU CLIENT
              2 . MODIFIER LOCALITE CLIENT
              3 . SUPPRIMER CLIENT
              4 . LISTE DES CLIENTS D''UNE CATEGORIE
              0 . FIN
.SELECT
 CODE 1, CHAIN 'PROG1'
 CODE 2, CHAIN 'PROG2'
 CODE 3, CHAIN 'PROG2'
 CODE 4, CHAIN 'PROG3'
 CODE 0, STOP
.END
.OSPEND-MENUP
.SL ITS
   05 ITS-CHOIX PIC X.
.SL NEXT
   MOVE DCV-FUNCTION TO ITS-CHOIX.
.OSPEND
```

```
**PDL*9007061629/PROG1/00/
.PROG-PROG1,PROGMODE=TRANSACTION
.OSP,DATA-ENTRY,RD RDCLIENT
.OSP-ECRENR,LAYOUT-DESCRIPTION ECRENRMC
.OSPEND-ECRENR
.FILE-CLIENTS,DECLARE,-
.     FD=FDCLIENT
.FILEND
.FILE-CLI2,INPUT,-
.     FD=FDCLIENT,-
.     FN=CLIENTS
.FILEND
.SL WORK01
   01 NOUVNUM  PIC 9(5).
.SL INITIATE
.SPP
   GET CLI2.
   DO TRT WHILE STATIN-CLI2 NOT = EOF.
        MOVE CLI2-NUMERO TO NOUVNUM.
        GET CLI2.
   END TRT.
.END
.FILE-CLI1,UPDATE-ONPLACE,-
.    FD=FDCLIENT,-
.    FN=CLIENTS,-
.    KEY-1=(ITS-NUMERO,9(5))
.FILEND
.SL PUTDB
.SPP
   COMPUTE NOUVNUM = NOUVNUM + 1.
   MOVE "N" TO ITS-CATEGORIE.
   MOVE NOUVNUM TO ITS-NUMERO.
   MOVE ITS-REC TO CLI1-REC.
   MOVE NEW TO STATOUT-CLI1.
   PUT CLI1.
.END
.OSPEND
```

```
**PDL*9007061901/PROG2/00/
.PROG-PROG2,PROGMODE=TRANSACTION
.OSP,UPDATE,KEY-1=(NUMERO,N,5,MUST),-
.     RD RDCLIENT,-
.     KEYCHANGE=EXIT
.OSP-ECRMAJ,LAYOUT-DESCRIPTION=ECRMAJMC
.OSPEND-ECRMAJ
.SL GETRDB
.SPP
       GETR CLIENTS.
       IF STATIN-CLIENTS =  DC-NOTFOUND.
          .DCERROR, 'Client inexistant...', CURSOR NUMERO
       END.
       MOVE STATIN-CLIENTS TO STATIN-DB.
       MOVE CLIENTS-REC TO ITS-REC.
.END
.SL PUTDB
.SPP
     SELECT STATUS-DB.
       CASE = DC-STORE.
            MOVE REPL TO STATOUT-CLIENTS.
       CASE = DC-DEL.
            MOVE DEL TO STATOUT-CLIENTS.
       ELSE.
            MOVE OLD TO STATOUT-CLIENTS.
       END.
      MOVE ITS-REC TO CLIENTS-REC.
      PUT CLIENTS.
.END
.OSPEND
.FILE-CLIENTS,UPDATE-ONPLACE,-
.   FD FDCLIENT,-
.   FN=CLIENTS,-
.   ACCESS-KEY=NUMERO,-
.   KEY-1=(DI-KEY,9(5))
.FILEND
```

```
**PDL*9007061437/PROG3/00/
.PROG-PROG3,PROGMODE=TRANSACTION
.OSP,BROWSE-REFERENCE,-
.   PAGES=2,LINES=5
.OSP-ECRLIS,LAYOUT-DESCRIPTION=ECRLISMC
.SEARCH
   -NUMERO,9(5)
.VIA
   BR-CATEGORIE, X,  CLIENTS
.SHOW
   -NUMERO,          9(5)
   -NOM,             X(30)
   -LOCALITE,        X(30)
.END
.OSPEND-ECRLIS
.FILE-CLIENTS,INPUT,-
.       FD FDCLIENT,-
.       ACCESS-KEY=CATEGORIE
.FILEND
.OSPEND
```

```
**PDL*9007060945/ECRAN/00/
.LAYOUT-ECRAN

                &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
                &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

      &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
      &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
      &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
      &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
      &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

                           CHOIX: &

.VARIABLES,PROTECTED
 TABLE-HEADER, TIMES 2
   -TITLE, BRIGHT
 TABLE-HEADER-END
 TABLE-INFOS, TIMES 5
   -ITEMTEXT, NORMAL
 TABLE-INFOS-END
   -FUNCTION, UNPROTECTED
.END
```

```
**PDL*9007051747/ECRENREG/00/
.LAYOUT-ECRENR, CURSOR NOM

              ENREGISTREMENT CLIENT :
              ======================

              NOM : &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

              LOCALITE : &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

              CHOIX :&

.VARIABLES,PREFIX ITS,UNPROTECTED,BRIGHT
   -NOM,UPDATE,MUST
   -LOCALITE,UPDATE,MUST
   -FUNCTION
.END
```

```
**PDL*9007061205/ECRMAJ.S/00/
.LAYOUT-ECRMAJ

            SUPPRESSION - MODIFICATION :
            ============================


            NUMERO : _____

            NOUVELLE LOCALITE : &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

            CHOIX : &

.VARIABLES,PREFIX=ITS
 -NUMERO,UPDATE
 -LOCALITE,UPDATE
 -FUNCTION
.END
```

```
**PDL*9007061438/ECRLIS.S/00/
.LAYOUT-ECRLIS
                                                          PAGE : _

   NUMERO NOM                              LOCALITE                          CAT
                                                                              &
 ==========================================================================
 __  ______ &&&&&&&&&&&&&&&&&&&&&&&&&&&&& &&&&&&&&&&&&&&&&&&&&&&&&&&&&& &
 __  ______ &&&&&&&&&&&&&&&&&&&&&&&&&&&&& &&&&&&&&&&&&&&&&&&&&&&&&&&&&& &
 __  ______ &&&&&&&&&&&&&&&&&&&&&&&&&&&&& &&&&&&&&&&&&&&&&&&&&&&&&&&&&& &
 __  ______ &&&&&&&&&&&&&&&&&&&&&&&&&&&&& &&&&&&&&&&&&&&&&&&&&&&&&&&&&& &
 __  ______ &&&&&&&&&&&&&&&&&&&&&&&&&&&&& &&&&&&&&&&&&&&&&&&&&&&&&&&&&& &
 ==========================================================================
 __          FONCTION &&                     PAGE SUIVANTE : __/__

.VARIABLES,PREFIX ITS
 -ACTPAGE,PROTECTED
 -BR-CATEGORIE,
 TABLE-CLIENTS,TIMES 5
 -SEL-CODE,PROTECTED
 -NUMERO,PROTECTED
 -NOM,PROTECTED
 -LOCALITE,PROTECTED
 -CATEGORIE,PROTECTED
 TABLE-CLIENTS-END
 -SELECTION,PROTECTED
 -FUNCTION,UNPROTECTED
 -NEXTPAGE,PROTECTED
 -TOTPAGES,PROTECTED
.END
```

Réalisation, impression et reliures : SPRL Centre de Reprographie - Tél. : 081/22.87.72